PATENT ABSTRACTS OF JAPAN

(11)Publication number : 2004-038928

(43)Date of publication of application : 05.02.2004

(51)Int.Cl. G06F 12/00
G06F 3/06

(21)Application number : 2003-075430

(22)Date of filing : 19.03.2003

(71)Applicant : NETWORK APPLIANCE INC

(72)Inventor : FEDERWISCH MICHAEL L
OWARA SHANE S
MANLEY STEPHEN L
KLEIMAN STEVEN R

(30)Priority

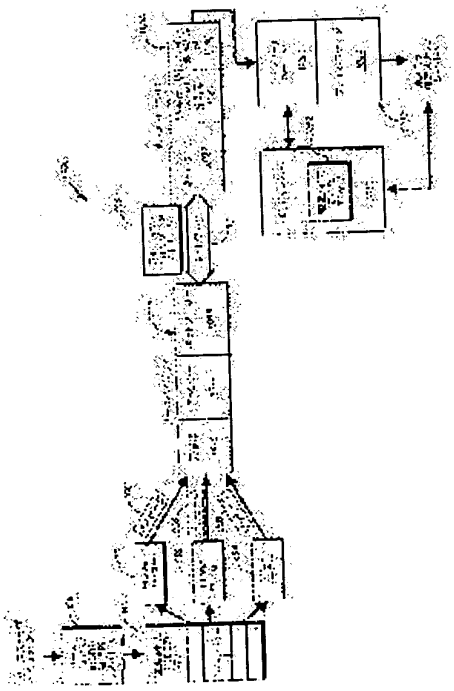
Priority number : 2002 100967 Priority date : 19.03.2002 Priority country : US
2002 100950 19.03.2002 US

(54) SYSTEM AND METHOD FOR DETERMINING CHANGE BETWEEN TWO SNAPSHOTS AND TRANSMITTING THE CHANGE TO DESTINATION SNAPSHOT

(57)Abstract:

PROBLEM TO BE SOLVED: To enhance versatility and utility of a snapshot duplication means.

SOLUTION: A system and a method are for performing remote asynchronous duplication, namely, mirroring of change of source file system snapshots to a duplicated file system at a destination by identifying a block changed by difference of block volume numbers identified in the case of scanning a logical file block index of each snapshot by utilizing scan (by a scanner) of blocks constituting two versions of snapshots of the source file system. A tree of the block regarding the file is checked, a pointer without changes between versions is bypassed and proceeded downward and changes of



hierarchy of trees are identified and the changes are transmitted to a destination mirror, namely, a duplicated snapshot. At the destination, the destination snapshot is updated by using source change. Every deleted or changed inode already existing on the destination is transferred to a temporary directory, namely, "momentary" directory and when it is reused, it is re-linked with the reconstructed duplicated snapshot directory.

LEGAL STATUS

[Date of request for examination]	16.09.2004
[Date of sending the examiner's decision of rejection]	
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]	
[Date of final disposal for application]	
[Patent number]	
[Date of registration]	
[Number of appeal against examiner's decision of rejection]	
[Date of requesting appeal against examiner's decision of rejection]	
[Date of extinction of right]	

(19)日本国特許庁(JP)　(12)公開特許公報(A)　(11)特許出願公開番号
特開2004-38928
(P2004-38928A)
(43)公開日　平成16年2月5日(2004.2.5)

(51)Int.Cl.⁷	F I		テーマコード（参考）
G06F 12/00	G06F 12/00	510B	5B065
G06F 3/06	G06F 12/00	501A	5B082
	G06F 12/00	531D	
	G06F 12/00	533J	
	G06F 3/06	304E	

審査請求 未請求　請求項の数 18　OL　外国語出願　（全 93 頁）

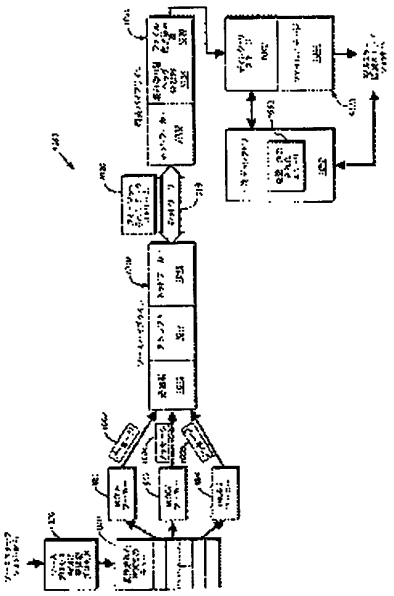
(21)出願番号　特願2003-75430(P2003-75430)
(22)出願日　平成15年3月19日(2003.3.19)
(31)優先権主張番号　10/100967
(32)優先日　平成14年3月19日(2002.3.19)
(33)優先権主張国　米国(US)
(31)優先権主張番号　10/100950
(32)優先日　平成14年3月19日(2002.3.19)
(33)優先権主張国　米国(US)

(71)出願人　500261341
ネットワーク・アプライアンス・インコーポレイテッド
アメリカ合衆国９４０８９カリフォルニア州サニーベイル、イースト・ジャーバ・ドライブ４９５番
(74)代理人　100087642
弁理士　古谷　聡
(74)代理人　100076680
弁理士　溝部　孝彦
(74)代理人　100121061
弁理士　西山　清春

最終頁に続く

(54)【発明の名称】２つのスナップショット間の変化を判定して宛先スナップショットに送信するシステム及び方法

(57)【要約】　（修正有）
【課題】スナップショット複製手段の汎用性及び有用性を向上させる。
【解決手段】ソースファイルシステムの２バージョンのスナップショットを構成するブロックのスキャン（スキャナによる）を利用して、各スナップショットの論理ファイルブロック索引のスキャンの際に識別されたブロックボリューム番号の差異により変更されたブロックを識別することで、ソースファイルシステムスナップショットの変化を宛先の複製ファイルシステムに遠隔非同期複製すなわちミラーリングするためのシステム及び方法。ファイルに関するブロックのツリーを調べ、バージョン間で変更のないポインタはバイパスして下方へ進み、ツリーの階層の変化を識別する。そしてこれらの変更を宛先ミラーすなわち複製スナップショットに送信する。宛先では、ソース変更を用いて宛先スナップショットを更新する。既に宛先上に存在するあらゆる消去または変更されたｉｎｏｄｅは、テンポラリディレクトリすなわち「一時」ディレクトリに移動され、再利用する場合、再構築さ



【特許請求の範囲】
【請求項１】
ソース上のデータブロックの論理グループにおける変更を識別し、宛先上のデータブロックの論理グループの複製を更新するためのシステムであって、
Ａ）第１のスナップショット及び第２のスナップショットそれぞれの第１のブロック識別子の集合及び第２のブロック識別子の集合に作用するスキャナであって、前記第１のスナップショット及び前記第２のスナップショットがデータブロックの論理グループの異なるイメージに対応し、前記第１のスナップショットが前記複製に対応し、前記第２のブロック識別子の集合の中から前記第１のブロック識別子の集合のうちの対応するブロック識別子と異なる識別子をサーチすることにより、前記複製に反映されていない変更されたデータブロックを示すようになっているスキャナと、
Ｂ）前記論理グループのブロックをすべて送信することなく、前記変更されたブロックを宛先に送信することと、
Ｃ）前記宛先上の複製を前記変更されたブロックで更新することと、
からなるシステム。
【請求項２】
前記スキャナは、変更のないボリュームブロック番号を有するブロックをバイパスしつつ、前記第１及び第２のスナップショットそれぞれの論理ブロック索引に関連付けられたポインタの階層を下方に移動してゆき、変更されたブロック識別子を有する指されているブロックを対応するオフセットにある指されているブロックに関して識別する、請求項１のシステム。
【請求項３】
所定の関係を有する検索されたブロックにおけるｉｎｏｄｅを取り出し、前記第１のスナップショット及び前記第２のスナップショットにおけるそれらのｉｎｏｄｅの第１のバージョンと第２のバージョンとの間の変化を示す、ｉｎｏｄｅ採集器をさらに含む、請求項１のシステム。
【請求項４】
前記所定の関係がボリュームファイルシステムの下位構成を含む、請求項３のシステム。
【請求項５】
前記下位構成前記ボリュームファイルシステム内のＱｔｒｅｅを含む、請求項４のシステム。
【請求項６】
前記ｉｎｏｄｅ採集器によって取得され待ち行列に入れられたブロックに対して作用し、取り出された前記ｉｎｏｄｅの各々の第１のバージョンの階層におけるブロック間の変化を、取り出された前記ｉｎｏｄｅの各々の第２のバージョンの階層に関して、前記スキャナを用いて計算する、ｉｎｏｄｅワーカーをさらに含む、請求項４のシステム。
【請求項７】
前記ｉｎｏｄｅワーカーは、前記変化を収集し、該変化をデータストリームにパッケージングするパイプラインに転送して、前記宛先ファイルシステムに伝送する、請求項６のシステム。
【請求項８】
前記ソースファイルシステムと前記宛先ファイルシステムとの間の変化を送信する拡張可能でファイルシステム非依存なフォーマットをさらに含む、請求項７のシステム。
【請求項９】
宛先上のデータブロックの論理グループの複製を、ソース上の対応する論理グループにおける変化を用いて更新するための方法であって、
Ａ）ソース上の論理グループの第１のスナップショットの第１のブロック識別子の集合が前記複製に対応し、これをソース上の論理グループの第２のスナップショットの第２のブロック識別子の集合と比較することにより、変更された論理グループ内の対応する位置にあるデータブロックを識別することと、

B）前記論理グループのブロックをすべて送信することなく、前記変更されたブロックを送信することと、
C）前記宛先上の複製を送信された前記変更されたブロックで更新することと、
からなる方法。
【請求項１０】
前記第１のスナップショット及び第２のスナップショットは第１及び第２の時点での前記ソース上の論理グループのイメージにそれぞれ対応し、前記比較するステップは、前記第１及び第２のブロック識別子の集合の対応するブロック識別子であって前記第１の時点と前記第２の時点との間で変更がなかったブロック識別子を選択する、請求項９の方法。
【請求項１１】
前記ソース上の論理グループは階層構造を含み、前記第１及び第２のブロック識別子の集合は、それぞれ前記階層構造に対応する階層的ブロック索引を含み、前記選択するステップは、前記階層的ブロック索引をサーチして変更されたブロックを識別するスキャナによって実施される、請求項１０の方法。
【請求項１２】
前記第１及び第２のｍブロック識別子の集合のブロック識別子はソース上の論理グループ内の対応する位置にあるデータブロックを参照し、前記選択するステップは、前記論理グループ内の同じ位置にあるデータブロックを参照する前記第１及び第２のブロック識別子の集合のブロック識別子を比較することにより、変更されたブロックを選択する、請求項１１の方法。
【請求項１３】
ｉｎｏｄｅ採集器を用いて所定の関係を有するｉｎｏｄｅを取り出し、前記第１のスナップショット及び前記第２のスナップショットにおけるそれらのｉｎｏｄｅの第１のバージョンと第２のバージョンとの間の変化を示すことをさらに含む、請求項９の方法。
【請求項１４】
前記所定の関係がボリュームファイルシステムの下位構成を含む、請求項１３の方法。
【請求項１５】
前記下位構成が前記ボリュームファイルシステム内のＱｔｒｅｅを含む、請求項１４の方法。
【請求項１６】
ソース上のデータブロックの論理グループから変更のデータストリームを受信して、宛先上の論理グループの複製を更新するためのシステムであって、
A）前記ソース上の論理グループの構造を表すメタデータを読み出し、ソース上の論理グループに対する参照を宛先上の論理グループに対する参照にマッピングして変更されたブロックを識別する、メタデータステージプロセスと、
B）マッピングされた前記参照に応じて、前記複製上の論理グループを、前記論理グループ内の対応するオフセットで変更のあった前記ソースからのデータブロックで埋める、データステージプロセスと、
からなるシステム。
【請求項１７】
ソース上のデータブロックの論理ブロックに対する変更を識別し、宛先上の前記論理グループの複製を更新するためのシステムであって、
A）前記ソース上の論理グループに関する構造を表すメタデータであって前記ソース上の論理グループのデータブロックに対する第１の参照の集合を含むメタデータを読み出すステップ（ｉ）と、読み出した前記メタデータの複製を生成するステップ（ｉｉ）とを含み、前記第１の参照の集合を宛先上の論理グループのデータブロックに対する第２の参照の集合にマッピングするステップを含む、メタデータステージプロセスと、
B）前記複製を前記第２の参照の集合によって参照されるデータブロックで埋めるステップを含み、前記メタデータステージプロセスに応答する、データステージプロセスと、
からなるシステム。

【請求項１８】
宛先上の複製を更新するための方法であって、
Ａ）スナップショットの変更されたデータから、宛先上で消去および変更されたデータの論理グループに関する識別子を読み出し、該消去および変更された論理グループを前記複製の主記憶装置とは別の一時記憶装置に入れることと、
Ｂ）前記一時記憶装置にある前記消去及び変更された論理グループに対する一連の参照を前記主記憶装置に作成することと、
Ｃ）前記作成するステップの後、前記主記憶装置にある前記消去及び変更されたデータの論理グループに対する参照を維持しつつ、前記一時記憶装置を再割り当てすることと、
からなる方法。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】
本発明はファイルサーバを用いたデータの記憶に関し、詳しくは遠隔の記憶場所に記憶されたデータのネットワークを介したミラーリングまたは複製に関する。
【０００２】
【従来の技術】
ファイルサーバは、ディスク等の記憶装置上での情報の編成に関するファイルサービスを提供するコンピュータである。ファイルサーバまたはファイラーには、ディスク上で情報をディレクトリ及びファイルの階層構造として論理的に編成するためのファイルシステムを実施するストレージオペレーティングシステムが含まれる。「ディスク上」の各ファイルは、ディスクブロック等のデータ構造の集まりとして実現され、情報を格納するように構成される。一方、ディレクトリは、他のファイル及びディレクトリに関する情報を格納する特別な形態のファイルとして実現される。
【０００３】
ファイラーは、クライアント／サーバ形態の情報配送に従って動作するようにさらに構成され、多数のクライアントがファイラー等のサーバ上に格納されたファイルにアクセスできるようになっている。この形態の場合、クライアントは、直接接続や、ポイント・ツー・ポイントリンク、共用ローカルエリアネットワーク（ＬＡＮ）、ワイドエリアネットワーク（ＷＡＮ）またはインターネット等の公衆網上で実施される仮想施設ネットワーク（ＶＰＮ）等のコンピュータネットワークを介してファイラーに「接続」するコンピュータ上で、データベースアプリケーション等のアプリケーションを実行している場合がある。各クライアントは、ネットワークを介してファイルシステムプロトコルメッセージを（パケットの形で）ファイラーに発行することによって、ファイルシステムのサービスをファイラーに要求することができる。
【０００４】
一般的なタイプのファイルシステムは「定位置書き込み」ファイルシステムであり、その一例はパークレイ・ファーストファイルシステムである。「ファイルシステム」はディスク等の記憶装置上のデータ及びメタデータの構造を一般に意味し、ファイルシステムによってそれらのディスクに対するデータの読み書きが可能になる。定位置書き込みファイルシステムの場合、ｉｎｏｄｅやデータブロック等のデータ構造のディスク上での位置が通常、固定になっている。ｉｎｏｄｅはファイルに関するメタデータ等の情報を記憶するのに用いられるデータ構造であり、データブロックはそのファイルの実際のデータを記憶するのに用いられるデータ構造である。ｉｎｏｄｅに保持される情報には、例えばそのファイルの所有者、そのファイルのアクセス権、そのファイルのサイズ、そのファイルのタイプ及びデータブロックのディスク上の位置に対する参照などが含まれる。ファイルデータの位置に対する参照は、ポインタとしてｉｎｏｄｅに設けられ、そのファイルのデータ量に応じて、間接ブロックの参照、即ちデータブロックへの参照をさらに含む場合もある。ｉｎｏｄｅ及びデータブロックに対する変更は、固定位置書き込みファイルシステムに従って「定位置」で行なわれる。ファイルの更新がそのファイルのデータ量を超えるもので

ある場合、さらにデータブロックが割り当てられ、そのデータブロックを参照するように適当なｉｎｏｄｅが更新される。

【０００５】

他のタイプのファイルシステムとしては、ディスク上のデータを上書きしないＷｒｉｔｅ－ａｎｙｗｈｅｒｅファイルシステムがある。ディスクからそのディスク上のデータブロックがメモリに取得され（読み出され）、新たなデータで汚される場合、そのデータブロックがディスク上の新たな位置に格納される（書き込まれる）ので、書き込み効率が最適になってる。Ｗｒｉｔｅ－ａｎｙｗｈｅｒｅファイルシステムは、データがディスク上で実質的に連続的に配置されるように、まず最適なレイアウトを推定する。この最適なレイアウトによって、特にディスクに対するシーケンシャル読み出し処理に関して、効率的なアクセス処理が可能になる。ファイラー上で動作するように構成されたＷｒｉｔｅ－ａｎｙｗｈｅｒｅファイルシステムの一例は、カリフォルニア州サニーベイルにあるネットワークアプライアンス社から入手できるＷｒｉｔｅ Ａｎｙｗｈｅｒｅ Ｆｉｌｅ Ｌａｙｏｕｔ（ＷＡＦＬ）ファイルシステムである。ＷＡＦＬファイルシステムは、ファイラー及び関連ディスクストレージのプロトコルスタック全体のうちの一部としてマイクロカーネル内で実施される。このマイクロカーネルは、ネットワークアプライアンス社のＤａｔａ ＯＮＴＡＰストレージオペレーティングシステムの一部として提供され、ネットワーク取り付けされたクライアントからのファイルサービス要求を処理する。

【０００６】

本明細書で用いられるように「ストレージオペレーティングシステム」という用語はデータアクセスを管理するコンピュータ上で実行可能なコンピュータ実行可能コードを意味し、ファイラーの場合、このコードはカリフォルニア州サニーベイルにあるネットワークアプライアンス社から入手可能なＤａｔａ ＯＮＴＡＰ（Ｒ）ストレージオペレーティングシステム等のファイルシステムセマンティックを実施するものであり、Ｄａｔａ ＯＮＴＡＰストレージオペレーティングシステムはマイクロカーネルとして実施され、Ｗｒｉｔｅ Ａｎｙｗｈｅｒｅ Ｆｉｌｅ Ｌａｙｏｕｔ（ＷＡＦＬ（Ｒ））ファイルシステムを実現する。ストレージオペレーティングシステムは、ＵＮＩＸ（Ｒ）やＷｉｎｄｏｗｓ（Ｒ） ＮＴ等の汎用オペレーティングシステム上で動作するアプリケーションプログラムとして実施することもできるし、本明細書に記載するようなストレージアプリケーションとして構成された設定可能な機能を有する汎用オペレーティングシステムとして実施することもできる。

【０００７】

ディスクストレージは通常、ストレージ空間の全体的な論理的配置を定義する、物理的記憶ディスクを含む１以上のストレージ「ボリューム」として実施される。現在利用可能なファイラー実施形態は、多数の個別のボリューム（例えば１５０以上）を使用することができる。各ボリュームにはそのボリュームに固有のファイルシステムが関連付けられ、その意味で、ボリュームとファイルシステムは一般に同義語として用いられる。ボリューム内のディスクは通常、１以上のグループのＲＡＩＤ（Ｒｅｄｕｎｄａｎｔ Ａｒｒａｙ ｏｆ Ｉｎｄｅｐｅｎｄｅｎｔ（Ｉｎｅｘｐｅｎｓｉｖｅ） Ｄｉｓｋｓ）に編成される。ＲＡＩＤ実施形態は、ＲＡＩＤグループ内の所定数の物理ディスクにわたってデータ「ストライプ」を冗長書き込みし、そのストライプ化されたデータに関するパリティ情報を適当にキャッシュすることによって、データ記憶の信頼性／完全性を向上させている。ＷＡＦＬファイルシステムの一例では、ＲＡＩＤ４実施形態が有利に用いられている。この実施形態では、１グループのディスクにわたってデータをストライピングし、ＲＡＩＤグループのうちの選択されたディスク内で個別にパリティをキャッシュする必要がある。本明細書に記載するように、ボリュームは通常、ＲＡＩＤ４または同等の高信頼性の実施形態に従って配置された、少なくとも１台のデータディスク及び１台の関連パリティディスク（データ／パリティの場合もある）パーティションを含む。

【０００８】

ファイラー、ファイラーに関連するディスク、または、ストレージインフラストラクチャ

の何らかの部分に故障が発生したときの信頼性を向上させ、災害普及を容易にするため、基礎的データ及び／又は該データを編成するファイルシステムのうちのいくらか又は全部を「ミラーリング」すなわち複製することが一般的である。例えばミラーは、離れたサイトに作成及び格納され、主要な格納場所やそのインフラストラクチャに真の災害（例えば洪水、停電、戦争活動など）が発生した際の復元の可能性を向上させる。ミラーは、ファイルシステムに対する最新の変更を把握するため、管理者によって設定された所定の時間間隔で更新されるのが一般的である。更新の一般的な一形態では、ｉｎｏｄｅとブロックで構成されるストレージサイトのアクティブファイルシステムを取り込み、ネットワーク（周知のインターネット等）を介して概して「スナップショット」を遠隔のストレージサイトに送信する、「スナップショット」プロセスを使用する。一般に、スナップショットは、ある時点でのファイルシステムのイメージ（通常は読み出し専用）であり、アクティブファイルシステムと同じ一次記憶装置に格納され、アクティブファイルシステムのユーザがアクセスできるようになっている。「アクティブファイルシステム」は、現在の入出力処理が向けられている対象となるファイルシステムを意味する。一連のディスク等の一次記憶装置はアクティブファイルシステムを格納するものであり、テープドライブ等の二次記憶装置はアクティブファイルシステムのバックアップを格納するために用いられる。

【０００９】

スナップショットを作成した後、起こり得る災害復旧に備えてその時点で作成されたスナップショットを残し、アクティブファイルシステムが再確立される。スナップショットを作成するたびに古いアクティブファイルシステムが新たなスナップショットになるので、新たなアクティブファイルシステムは何らかの変更を記録し続ける。様々な時間ベースの基準及びその他の基準に応じて、所定数のスナップショットが維持される。スナップショットを作成する処理の詳細については、「ＩＮＳＴＡＮＴ　ＳＮＡＰＳＨＯＴ」と題したＢｌａｋｅ　Ｌｅｗｉｓ他による米国特許出願第０９／９３２５７８号明細書に記載されており、この出願はここで参照することにより完全に説明したものとして本明細書に取り込まれる。また、ＷＡＦＬファイルシステムに固有のＳｎａｐｓｈｏｔ（Ｒ）機能については、ネットワークアプライアンス社が発行している「Ｆｉｌｅ　Ｓｙｓｔｅｍ　Ｄｅｓｉｇｎ　ｆｏｒ　ＮＦＳ　Ｆｉｌｅ　Ｓｅｒｖｅｒ　Ａｐｐｌｉａｎｃｅ」と題したＤａｖｉｄ　Ｈｉｔｚ他によるＴＲ３００２、及び、同所有者の「ＭＥＴＨＯＤ　ＦＯＲＭＡＩＮＴＡＩＮＩＮＧ　ＣＯＮＳＩＳＴＮＴ　ＳＴＡＴＥＳ　ＯＦ　Ａ　ＦＩＬＥ　ＳＹＳＴＥＭ　ＡＮＤ　ＦＯＲ　ＣＲＥＡＴＩＮＧ　ＵＳＥＲ－ＡＣＣＥＳＳＩＢＬＥ　ＲＥＡＤ－ＯＮＬＹ　ＣＯＰＩＥＳ　ＯＦ　Ａ　ＦＩＬＥ　ＳＹＳＴＥＭ」と題したＤａｖｉｄ　Ｈｉｔｚ他による米国特許第５８１９２９２号明細書にさらに詳しい説明があり、これらはここで参照することにより取り込まれる。

【００１０】

ファイルシステムの大きさが数十ギガバイトや数百ギガバイト（数テラバイトと同じ）に及ぶ場合、ネットワークを介してファイルシステム全体をリモート（宛先）サイトに完全に複製することは、かなり不便である。遠隔地にデータ複製するためのこの完全バックアップ方法は、ネットワークの帯域幅に激しい負担をかける場合があり、複製先と複製元の両方ファイラーの処理能力にも負担をかける場合がある。一解決方法として、スナップショットをファイルシステムボリュームのうちの変更を受けた部分のみに制限するということが行なわれている。従って、図１は従来技術によるボリュームベースのミラーリングを示すものであり、この図において、複製元ファイルシステム１００はネットワークリンク１０４を介して宛先記憶サイト１０２（サーバとそれに取り付けられた記憶装置で構成される。図示せず）に接続されている。宛先１０２は、管理者によって設定された所定の時間間隔で周期的にスナップショットの更新を受信する。これらの時間間隔は、帯域幅、データの重要性、変更の頻度、及び、ボリューム全体のサイズ等を含む様々な基準をもとにして選択される。

【００１１】

簡単にまとめると、ソースは、時間をあけて、そのボリュームの１対のスナップショット

を作成する。これらのスナップショットは、データをファイラーの不揮発性メモリに送信する送信処理の部分で作成することができ、あるいは他の手段によって作成することもできる。「新たな」スナップショット１１０は、そのボリュームのアクティブファイルシステムの最新のスナップショットである。「古い」スナップショット１１２は、そのボリュームの古いスナップショットであり、宛先のミラーに複製されているファイルシステムイメージに一致するはずのものである。注意して欲しいのは、ファイルサーバは、新たなスナップショット１１２の作成が終われば、ファイルサービス要求に対する作業を自由に続けられるということである。この新たなスナップショットは、現在のボリューム状態を完全に表現するものではなく、その時刻に至るまでの活動状態のチェックポイントとして働くものである。相違検出器１２０は古いスナップショットと新たなスナップショットを調べる。具体的には、相違検出器は、各スナップショットのブロックのリストを１ブロックづつ検査して、そこに配置されているブロック同士を比較する。Ｗｒｉｔｅ－ａｎｙｗｈｅｒｅファイルシステムの場合、スナップショットがブロックを参照している限りそのブロックは再利用されないので、データの変更は新たなブロックに書き込まれる。変更が確認された場合（データを意味する「×」の存在または不在で図示する）、相違検出器１２０は、図２に示す判断処理２００によってそのデータを宛先１０２へ送信するか否かを判定する。この判断処理２００は、古いブロックと新しいブロックとを次のような手順で比較する。（ａ）新／旧ブロック対１３０のように古いブロックにも新しいブロックにもデータがない場合（２００の場合）、転送可能なデータはない。（ｂ）新／旧ブロック対１３２のように古いブロックにはデータがあるが新しいブロックにはデータがない場合（２０４の場合）、そのようなデータは転送が済んでいるので（新たな宛先スナップショットポインタはいずれもそのデータを無視するので）、新たなブロック状態は転送しない。（ｃ）新／旧ブロック対１３４のように古いブロックと新たなブロックとの両方にデータが存在する場合（２０６の場合）、何も変更がなされていないので、そのブロックデータは既に前のスナップショットに転送されている。（ｄ）最後に、新／旧ブロック対１３６のように古いブロックにはデータがないが新たなブロックにはデータがある場合（２０８の場合）、変更されたデータブロックをネットワークを介して転送し、変更されたブロック１４２のように、宛先の変更されたボリュームスナップショット集合１４０の一部になるようにする。例示のＷｒｉｔｅ－ａｎｙｗｈｅｒｅファイルシステム構成の場合、これらの変更されたブロックは、ストレージアレイの新たな未使用の位置に書き込まれる。変更されたブロックがすべて書き込まれると、基礎ファイルシステム情報ブロック、すなわち、新たなスナップショットのルートポインタが宛先に送信される。このファイルシステム情報ブロックが宛先に送信されると、前のファイルシステム情報を置き換えて変更されたブロック構造を指し示すようにすることにより、宛先のファイルシステム全体が更新される。この時点で、変更が宛先ボリュームスナップショットの最新の差分更新として送信される。このファイルシステムは、ソースにある「新たな」スナップショットを正確に表現する。その後、さらなる変更差分から新しい「新たな」スナップショットが作成される。

【００１２】

ボリュームベースでスナップショットの遠隔ミラーリングを行なう方法については、同所有者の「ＦＩＬＥ　ＳＹＳＴＥＭ　ＩＭＡＧＥ　ＴＲＡＮＳＦＥＲ」と題したＳｔｅｖｅｎ　Ｋｌｅｉｍａｎ他による米国特許出願第０９／１２７４９７号明細書、及び、Ｓｔｅｎ　Ｋｌｅｉｍａｎ他による「ＦＩＬＥ　ＳＹＳＴＥＭ　ＩＭＡＧＥ　ＴＲＡＮＳＦＥＲ　ＢＥＴＷＥＥＮ　ＤＩＳＳＩＭＩＬＡＲ　ＦＩＬＥ　ＳＹＳＴＥＭＳ」と題した米国特許出願第０９／４２６４０９号明細書に詳しい説明があり、これら両方がここで参照することにより本明細書に取り込まれる。

【００１３】

このボリュームベースの方法は、ソースから遠隔の記憶先へ漸進的にミラーリングするのに有効ではあるが、ボリューム全体についての変更を検査して、それらの変更を１ブロックづつ送信しなければならないという点で、非効率で時間のかかる方法である。言い換えると、この検査は、ブロックに含まれるファイル、ｉｎｏｄｅ及びデータ構造に関する基

破情報を何も考慮することなく、ブロックに着目して行われる。宛先が一意のボリュームに編成されているので、ソースと宛先との間には、直接ボリューム単位のマッピングが確立される。ここでも、１ボリュームは１テラバイト以上もの情報を含む場合があり、変更を検査して比較する１ブロックづつの方法は、相当なプロセッサオーバヘッドとそれに関連する処理時間を必要とする。多くの場合、検査中のルートｉｎｏｄｅブロックの下方にあるサブブロックにわずかな変更が施されているだけである。しかしながら、そのボリューム内のあらゆるブロックのリストを検査しており、ブロックのグループ分け（ファイル、ｉｎｏｄｅ構造など）の多くに変更がないという事実は考慮されていない。さらに、ボリューム全体のサイズ及び範囲が大きくなるにつれて、グループによっては頻繁に変更されるグループもあり、それらのグループの複製は、あまり頻繁に変更されない他のグループの複製よりも頻繁に更新することが望ましいので、ミラーリングしているデータはサブグループに分割することが極めて望ましい。さらに、元のサブグループと複製された（スナップショットの）サブグループとを１つのボリュームに混在させ、ボリューム全体を移動させることなく特定の重要データを遠隔地に移動することが望ましい。従って、全体よりも少ないボリュームのミラーリングを可能にするボリュームの下位編成と、変更されたブロックをスキャンして特定するための、もっと洗練された方法が望まれている。

【００１４】

このようなボリュームの下位編成の１つが、周知のＱｔｒｅｅである。Ｑｔｒｅｅは、本明細書に記載される例示的ストレージシステム等で実施されるように、ボリュームのファイルシステムにおけるサブツリーである。Ｑｔｒｅｅの重要な特徴は、あるＱｔｒｅｅが与えられると、そのＱｔｒｅｅへの所属についてシステム内のあらゆるファイル及びディレクトリを迅速に検査することができるので、Ｑｔｒｅｅはファイルシステムを個別のデータ集合に編成するための優れた手段として機能するということである。スナップショットを作成するデータのソース及び宛先として、Ｑｔｒｅｅを使用することが望ましい。

【００１５】

【発明が解決しようとする課題】

宛先に複製されたスナップショットがＱｔｒｅｅであれば、そのスナップショットは、災害復旧やデータ配布等、他の用途にも利用することができる。しかしながら、ユーザまたはプロセスがアクセスしようとしたとき、宛先のアクティブファイルシステム上にあるスナップショットは、ソーススナップショットを受信中であったり、ソーススナップショットからの更新を処理中であったりする場合がある。干渉されることなくスナップショットの更新を完了できるようにする方法が強く望まれている。同様に、スナップショットを初期の状態に戻す必要がある場合、そのような復元すなわち「巻き戻し」を容易にする方法も望まれている。様々な時点における複数のスナップショットを処理するための様々なその他の技術は、スナップショット複製手段の汎用性及び有用性を向上させるものである。

【００１６】

さらに、宛先スナップショットが更新される速度は、変更データをソースから宛先のアクティブファイルシステムへ送信可能な速度にも、依存する場合がある。ファイル消去及びファイル変更の効率を向上させる方法も望まれている。

【００１７】

【課題を解決するための手段】

本発明は、ソースファイルシステムの２バージョンのスナップショットを構成するブロックのスキャン（スキャナを介した）であって、各スナップショットの論理ファイルブロック索引をスキャンした際に特定されるボリュームブロック番号の差異に基づいて各々のスナップショットにおける変更されたブロックを特定するスキャンを利用して、ソースファイルシステムスナップショットにおける変更を宛先複製ファイルシステムに遠隔非同期複製すなわちミラーリングするためのシステム及び方法を提供することにより、従来技術の欠点を克服する。バージョン間で変更のなかったポインタをバイパスし、Ｔｒｅｅの階層における変更を識別するように下方へ移動しながら、ファイルに関するブロックのＴｒｅ

ｅの中を移動する。これらの変更は、宛先のミラーすなわち複製スナップショットに送信される。この方法によって、通常のファイル、ディレクトリ、ｉｎｏｄｅ及び任意の他の階層構造を効率的にスキャンし、それらのバージョン間の差異を判定することが可能になる。

【００１８】

例示的実施形態によると、各スナップショットに対する論理ファイルブロックの索引に沿ったスキャナを用いたこのソーススキャンは、２つのソーススナップショット間で変更のあったボリュームブロック番号を探し出す。ディスクブロックが常にディスク上の新たな位置に再書き込みされるので、差異は、個々のブロックの基礎とするｉｎｏｄｅの変化を示す。スキャナを用いると、未変更のブロックは、それらのｉｎｏｄｅが変更されていないので、有効に見過ごされる。変更のあったブロックをスキャナから受信するｉｎｏｄｅ採集器プロセスを用いて、ソースは、選択されたＱｔｒｅｅ（または他のボリューム下位構成）に特に関連する変更されたブロックからｉｎｏｄｅを取り出す。この採集器プロセスは、２つのスナップショット間でバージョンが変更されたｉｎｏｄｅを探し出し、その変更されたバージョンを取り出す。ｉｎｏｄｅは同じであるがファイルには変更が施されている場合（ｉｎｏｄｅの生成番号が異なることで分かる）、それぞれのｉｎｏｄｅの２つのバージョンを両方とも取り出す。一方のバージョンで変更のあったｉｎｏｄｅは他方のバージョンでそれらのｉｎｏｄｅ以外のデータブロックを指し示すので、バージョンが変更された（２つのスナップショット間で）ｉｎｏｄｅは、キューに入れられ、一連のｉｎｏｄｅハンドラ／ワーカーに転送され、該ハンドラが、ｉｎｏｄｅの「ツリー」の下方へ向かってファイルオフセットをスキャンし（これもスキャナを用いて）、基礎とするブロックにおける差異がそれらのブロックのブロックポインタによって識別されるまでスキャンを続けることにより、基礎とするブロックにおける変化を解決する。非同期（遅延）方法の場合、宛先ファイルシステムの更新のためにネットワークを介して送信されるのは、ツリーの変更だけである。宛先ファイルシステムは、読み出し専用でユーザにエクスポートされる。これによって、複製者だけしか複製ファイルシステムの状態を変更できないことが保証される。

【００１９】

例示的実施形態では、変更データのデータストリームのネットワークを介した送信には、ファイルシステム非依存のフォーマットが用いられる。このフォーマットは、一意の識別子を有する単独ヘッダの集合で構成される。ヘッダが後続のデータを指し示す場合もあるし、ストリーム内に関連データが含まれる場合もある。例えば、「消去ファイル」ヘッダの内部には、ソーススナップショットで消去されたファイルに関する情報が含まれる場合がある。先ずディレクトリ活動がすべて送信され、その後ファイルデータが送信される。ファイルデータは、終端ヘッダ（フッタ）が設定されるまで規定のヘッダで分離された様々なサイズの塊で送信される。宛先では、このフォーマットをアンパックし、その中に含まれるｉｎｏｄｅをネットワーク上に送信し、それらを新たなディレクトリ構造にマッピングする。受信されたファイルデータブロックは、それらのオフセットに従って対応する宛先ファイルに書き込まれる。ｉｎｏｄｅマップは、ソースのｉｎｏｄｅ（ファイル）を宛先のｉｎｏｄｅ（ファイル）にマッピングするマップを格納する。ｉｎｏｄｅマップは、生成番号も保持する。（ｉｎｏｄｅ番号、生成番号）のタプルによって、システムは、ファイルに対して高速アクセスするためのファイルハンドルを作成することが可能になる。また、このタプルによってシステムは、ファイルが消去された際の変化を追跡し、そのｉｎｏｄｅ番号を新たに作成したファイルに再割り当てすることが可能になる。宛先での新たなディレクトリツリーの作成を容易にするため、宛先ミラープロセスの最初のディレクトリステージは、前記フォーマットでソースディレクトリ情報を受信し、消去されたファイルまたは移動されたファイルをすべてテンポラリなすなわち「一時的な」ディレクトリに移動する。移動されたそれらの一時的ファイルは、一時的ディレクトリからそれらのファイルが移動された先のディレクトリにハードリンクされる。新たに作成されたソースファイルがマップに入れられ、そのディレクトリツリーに組み込まれる。ディレクトリツ

リーが作成された後、ファイルデータの転送を開始する。ソースからのファイルデータに対する変更は、対応する複製ファイル（ｉｎｏｄｅマップによって識別される）に書き込まれる。データストリーム転送が完了すると、一時的ディレクトリを消去し、リンク先のないあらゆるファイル（様々な消去されたファイルを含む）を永久に消去する。一実施形態では、複数の個別のソースＱｔｒｅｅ、または、異なるソースボリュームから作成された他の下位組織を、１つの宛先ボリューム上に複製／ミラーリングすることができる。

【００２０】

本発明は、複製された宛先ファイルシステムスナップショットをソースファイルシステムスナップショットの変化分を用いて更新するためのシステム及び方法において、宛先アクティブファイルシステム上で変更されたファイル及び消去されたファイルをそれらのファイルが再利用されるまでずっと一時的ディレクトリに関連付けておくことを可能にするテンポラリすなわち「一時的」ディレクトリを用いて、宛先上でのソース更新情報からの新たなディレクトリツリーの作成を容易にすることによって、従来技術の欠点を克服する。さらに、ソースｉｎｏｄｅ番号から宛先ｉｎｏｄｅ番号へのマッピングを行なうｉｎｏｄｅマップを宛先上に設けることで、ｉｎｏｄｅ／生成番号のタプルを用いた宛先ツリーの作成を容易にしている。このｉｎｏｄｅマップにより、ソースファイルシステムと宛先との再同期が可能になる。また、このｉｎｏｄｅマップによって、２以上の宛先スナップショットを、それら各々のソースを有するマップに基いて、互いに関連付けることが可能になる。

【００２１】

例示的実施形態では、ファイルシステム非依存のフォーマットを用いて、ソースの基礎スナップショット及び差分スナップショットに関して変更のあったファイルデータブロックのデータストリームが送信される。受信したファイルデータブロックは、それらのオフセットに従って対応する宛先ファイルに書き込まれる。ｉｎｏｄｅマップは、ソースのｉｎｏｄｅ（ファイル）を宛先のｉｎｏｄｅ（ファイル）にマッピングするマップを格納する。ｉｎｏｄｅマップは、生成番号も保持する。（ｉｎｏｄｅ番号、生成番号）のタプルによって、システムは、ファイルに対して高速アクセスするためのファイルハンドルを作成することが可能になる。また、このタプルによってシステムは、ファイルが消去された際の変化を追跡し、そのｉｎｏｄｅ番号を新たに作成したファイルに再割り当てすることが可能になる。宛先での新たなディレクトリツリーの作成を容易にするため、宛先ミラープロセスの最初のディレクトリステージは、前記フォーマットでソースディレクトリ情報を受信し、消去されたファイルまたは移動されたファイルをすべてテンポラリなすなわち「一時的な」ディレクトリに移動する。移動されたそれらの一時的ファイルは、一時的ディレクトリからそれらのファイルが移動された先のディレクトリにハードリンクされる。新たに作成されたソースファイルがマップに入れられ、そのディレクトリツリーに組み込まれる。ディレクトリツリーが作成された後、ファイルデータの転送を開始する。ソースからのファイルデータに対する変更は、対応する複製ファイル（ｉｎｏｄｅマップによって識別される）に書き込まれる。データストリームの転送が完了すると、一時的ディレクトリを消去し、リンク先のないあらゆるファイル（様々な消去されたファイルを含む）を永久に消去する。一実施形態では、複数の個別のソースＱｔｒｅｅ、または、異なるソースボリュームから作成された他の下位構成を、１つの宛先ボリューム上に複製／ミラーリングすることができる。

【００２２】

別の例示的実施形態では、複製ファイルシステムそれ自体のスナップショットを作成することにより、第１のエキスポートされたスナップショットを作成する。この第１のエキスポートされたスナップショットは第１の状態に対応する。複製スナップショットがさらに変更または更新された後、天災や通信障害が発生した場合、複製ファイルシステムに対するそれ以上の変更を停止／凍結し、続いて、凍結された複製ファイルシステムから第２の状態を表す第２のエキスポートされたスナップショットを作成する。複製ファイルシステムは、第２の状態と第１の状態とのデータの差異を判定し、それらの変化分を用いて第１

の状態を再現することにより、第２の状態から第１の状態に「巻き戻す」ことができる。
【００２３】
さらに別の例示的実施形態では、ソーススナップショットから転送されたｉｎｏｄｅを、ｉｎｏｄｅマップを用いて宛先にある複製／ミラーファイルシステムのｉｎｏｄｅにマッピングし、ソースの状態と宛先の状態とを再同期させる。宛先は新たな「ソース」になり、古い「ソース」に対するｉｎｏｄｅマップの転送が新たな「宛先」にネゴシエートされる。受信した古いｉｎｏｄｅマップは、ソース上に格納され、反転手順によってアクセスされて、新たなソースのｉｎｏｄｅ数に等しいＮ個のｉｎｏｄｅを有する新たな宛先マップが生成される。次いで新たな宛先は、新たなソースエントリに関連する各宛先について、その格納されたソースマップからエントリを可能であれば作成する。関連する生成番号もマッピングされ、これによって必要なファイルアクセステーブルが生成される。新たなソース索引上で新たな宛先をもたないエントリにはすべてゼロエントリとしてマークが付される。反転ｉｎｏｄｅマップが完成すると、新たなソースはその変更されたデータで新しい宛先を更新できるようになる。
【００２４】
関連実施形態では、同じソースの２つの複製／ミラースナップショットが互いに鏡像関係になるようにすることができる。上記反転実施形態のように、新たな「ソース」（古い複製）がそのｉｎｏｄｅマップを宛先システムに転送する。次いで宛先システムは、２つのシステムのｉｎｏｄｅ間の関係を判定する。「連想」プロセスは、両方のｉｎｏｄｅマップを同時に調べる（例えば、ｉｎｏｄｅ番号単位で同時に）。このプロセスは、元のソースからの各ｉｎｏｄｅについて、ｉｎｏｄｅマップの各々から「宛先ｉｎｏｄｅ／生成」を取り出す。次いでこのプロセスは、新たなソースを新たなｉｎｏｄｅマップの適当なマップ索引として扱う。このプロセスは、新しいソース生成番号を、宛先ｉｎｏｄｅ／生成とともに格納する。
【００２５】
【発明の実施の形態】
Ａ．ネットワーク及びファイルサーバ環境
さらなる背景として、図３は、本発明に各々有利に用いられる、ソースファイルサーバ３１０及び宛先ファイルサーバ３１２からなる一対の相互接続されたファイルサーバを有するストレージシステム環境３００を示す略ブロック図である。説明上、このソースファイルサーバは１以上のソースボリューム３１４を管理するネットワークコンピュータであり、ソースボリューム３１４の各々がストレージディスク３６０のアレイ（以下で更に説明する）を有している。同様に、宛先ファイラー３１２も、ディスク３６０のアレイから構成される１以上の宛先ボリューム３１６を管理する。ソースファイルサーバと宛先ファイルサーバ、すなわちファイラーは、ローカルエリアネットワークや周知のインターネット等のワイドエリアネットワークで構成されるネットワーク３１８を介して接続される。各ファイラー３１０、３１２に含まれる適当なネットワークアダプタは、ネットワーク３１８を介した通信をサポートする。ここでも説明上、ソースファイラー及び宛先ファイラー３１０、３１２の各々における似たような構成要素は、類似の参照符号で示している。本明細書で用いられるように、「ソース」という用語は本発明の対象データの移動元の場所として大まかに定義され、「宛先」という用語はデータの移動先の場所として大まかに定義される。ネットワークで接続されたソースファイラー及び宛先ファイラーは本明細書で用いられるソース及び宛先の一例であり、ソース及び宛先は、ダイレクトリンク、すなわちループバック（ローカルなソースと宛先の間でデータストリームを伝達するための単一コンピュータ内部の「ネットワーク」構成）によってリンクされたコンピュータ／ファイラーである場合もあり、その場合ソースと宛先は同じファイラーになる。以下で更に説明するように、ソース及び宛先は、ボリュームのソース下位構成及びボリュームの宛先下位構成であるものとして広く解釈される。実際、少なくとも１つの特別な場合、ソース下位構成及び宛先下位構成は、様々な時点で同じものになる場合がある。
【００２６】

ネットワーク接続されたソースファイラー及び宛先ファイラーからなる一対の例の場合、各ファイラー３１０及び３１２には、スタンドアロンのコンピュータを含めて、任意のタイプの専用コンピュータ（例えばサーバ）または汎用コンピュータを用いることができる。ソースファイラー及び宛先ファイラー３１０、３１２の各々は、システムバス３４５で相互接続されたプロセッサ３２０、メモリ３２５、ネットワークアダプタ３３０及びストレージアダプタ３４０を含む。また各ファイラー３１０、３１２は、情報をディスク上でディレクトリ及びファイルの階層構造として論理編成するファイルシステムを実現するストレージオペレーティングシステム４００（図４）も含む。

【００２７】

当業者であれば、本明細書に記載する本発明の方法は、ストレージシステムとして実現されたスタンドアロンのコンピュータを含めて、任意のタイプの専用コンピュータ（例えばファイルサービス装置）または汎用コンピュータに適用できることが分かるであろう。そのため、ファイラー３１０及び３１２は各々、大まかに、及び代替として、ストレージシステムと呼ばれる場合もある。また、本発明の教示は様々なストレージシステムアーキテクチャに適用することができ、限定はしないがそれらのアーキテクチャには、ネットワークに取り付けられたストレージ環境、ストレージエリアネットワーク、及び、クライアント／ホストコンピュータに直接取り付けられたディスクアセンブリなどが含まれる。

【００２８】

例示的実施形態では、ソフトウェアプログラムコードを記憶するため、メモリ３２５には、プロセッサ及びアダプタによってアドレス指定可能な記憶場所が含まれる。メモリは、ランダムアクセスメモリ（ＲＡＭ）の形態に構成され、電源サイクルまたはその他リブート処理によってクリアされるのが通常である（すなわち、メモリは「揮発性」メモリである）。次に、プロセッサ及びアダプタは、そのソフトウェアコードを実行してデータ構造を操作するように構成された処理要素及び／又は論理回路で構成される。一般に、ストレージオペレーティングシステム４００は、その一部がメモリに存在し、処理要素によって実行され、とりわけファイラーが実施するファイルサービスをサポートする記憶処理を呼び出すことによって、ファイラーの機能を構成する。当業者であれば、本明細書に記載する本発明の方法に関するプログラム命令の格納及び実行には、様々なコンピュータ読み取り可能な媒体を含めて、他の処理手段及び記憶手段を用いることも可能であることが分かるであろう。

【００２９】

ネットワークアダプタ３３０は各ファイラー３１０、３１２をネットワーク３１８に接続するのに必要な機械回路、電気回路及び信号回路を含み、ネットワーク３１８はポイント・ツー・ポイント接続や、ローカルエリアネットワーク等の共用媒体からなる。さらに、ソースファイラー３１０は、クライアント／サーバモデルの情報配送に従って、宛先ファイラー３１２と対話することができる。従って、例えばＴＣＰ／ＩＰその他のネットワークプロトコル形態にカプセル化されたパケット３５５をネットワーク３１８を介して交換することにより、クライアントはファイラーのサービスを要求することができ、ファイラーはクライアントから要求されたサービスの結果を返すことができる。

【００３０】

ストレージアダプタ３４０は、ファイラー上で実行されているストレージオペレーティングシステム４００（図４）と協働して、クライアントから要求された情報にアクセスする。この情報は、ストレージアダプタ３４０を介して各ファイラーに取り付けられたディスク３６０上に格納されている場合もあるし、本明細書で定義されるようなストレージシステムの他のノードに格納されている場合もある。ストレージアダプタ３４０は、従来の高性能ファイバーチャネルシリアルリンク接続形態等、Ｉ／Ｏ相互接続構成によってディスクに接続するための入出力（Ｉ／Ｏ）インタフェース回路を有している。情報はストレージアダプタによって取得され、以下で詳しく説明するように、その情報をパケットに成形して宛先サーバへ送信する場合、システムバス３４５を介してネットワークアダプタ３３０に転送する前に、スナップショット手順の一部としてプロセッサ３２０によって処理さ

れる。

【００３１】

各ファイラーは、ネットワークアダプタ３３０を介して１以上のクライアント３７０と相互接続される場合もある。クライアントは、ファイルサービスの要求をソースファイラー及び宛先ファイラー３１０，３１２のそれぞれに送信し、それらの要求に対する応答をＬＡＮその他のネットワーク（３１８）を介して受信する。データは、クライアントと個々のファイラー３１０、３１２との間で、共通インターネットファイルシステム（ＣＩＦＳ）プロトコルまたはＮＦＳ等の他の適当なプロトコルのカプセル化として定義されたデータパケット３７４を用いて転送される。

【００３２】

一例示的なファイラー実施形態では、各ファイラー３１０，３１２は、データのフォールトトレランスなバックアップを可能にする不揮発性ランダムアクセスメモリ（ＮＶＲＡＭ）３３５を含む場合があり、ファイラートランザクションの完全性を確保して電源故障その他の故障によるサービス停止を切り抜けることができる場合がある。このＮＶＲＡＭのサイズは、ファイラーの実施形態及び機能に応じて決まる。通常は、特定時間（例えば数秒分）のトランザクションのログをとるのに十分なだけのサイズにする。各クライアント要求が完了した後、その要求の結果を要求しているクライアントに返す前に、ＮＶＲＡＭはバッファキャッシュと並行して満たされる。

【００３３】

一例示的実施形態では、ディスク３６０は複数のボリューム（例えばソースボリューム３１４及び宛先ボリューム３１６）に配置され、各ボリュームがそのボリュームに関連するファイルシステムを有するようになっている。これらのボリュームは、各々１以上のディスク３６０を有する。一実施形態では、好ましいＲＡＩＤ４構成に従って、物理ディスク３６０がＲＡＩＤグループに構成され、いくつかのディスクがストライプ化されたデータを格納し、いくつかのディスクがそのデータについて個別パリティを格納する場合がある。しかしながら、他の構成（例えばパリティが複数のストライプに分散されたＲＡＩＤ５など）も考えられる。この実施形態の場合、最小で、１台のパリティディスクと１台のデータディスクを使用する。しかしながら、典型的な実施形態では、ＲＡＩＤグループ１つ当たり３台のデータディスクと１台のパリティディスクが含まれ、１ボリューム当たり複数のＲＡＩＤグループが含まれる。

【００３４】

Ｂ．ストレージオペレーティングシステム

ディスク３６０に対する一般的なアクセスを容易にするため、ストレージオペレーティングシステム４００（図４）は、情報をディスク上でディレクトリ及びファイルの階層構造として論理編成するＷｒｉｔｅ－ａｎｙｗｈｅｒｅファイルシステムを実施する。「ディスク上」の各ファイルはデータ等の情報を格納するように構成されたディスクブロックの集まりとして実現され、ディレクトリは他のファイル及びディレクトリが格納された特別な形態のファイルとして実現される。上で説明及び定義したように、本明細書で説明する例示的実施形態の場合、ストレージオペレーティングシステムはカリフォルニア州サニーベイルにあるネットワークアプライアンス社から入手可能なＮｅｔＡｐｐ Ｄａｔａ ＯＮＴＡＰ（Ｒ）オペレーティングシステムであり、このオペレーティングシステムはＷｒｉｔｅ Ａｎｙｗｈｅｒｅファイルレイアウト（ＷＡＦＬ（Ｒ））ファイルシステムを実施する。任意の適当なファイルシステムを使用することが可能であるから、「ＷＡＦＬ」という用語を用いた場合であっても、この用語は本発明の教示に適合可能な任意のファイルシステムを意味するように広く解釈すべきである。

【００３５】

例示的ファイラーの各々について、好ましいストレージオペレーティングシステムの構成をまず簡単に説明する。しかしながら、明らかに、本発明の原理は様々な代替ストレージオペレーティングシステムアーキテクチャを用いて実施することもできると考えられる。さらに、ソースファイラー及び宛先ファイラー３１０、３１２の各々で実施される特定の

機能は、異なる場合もある。図４に示すように、例示的ストレージオペレーティングシステム４００は、ネットワークドライバ（例えばイーサネット（Ｒ）ドライバ）のメディアアクセス層を含む、一連のソフトウェア層から構成される。このオペレーティングシステムは更に、インターネット・プロトコル（ＩＰ）層４１０、並びに、ＩＰ層がサポートする搬送手段であるトランスポート・コントロール・プロトコル（ＴＣＰ）層４１５及びユーザ・データグラム・プロトコル（ＵＤＰ）層４２０を含む。ファイルシステムプロトコル層は、複数プロトコルのデータアクセスを可能にするため、ＣＩＦＳプロトコル４２５、ＮＦＳプロトコル４３０、及び、ハイパー・テキスト・トランスファー・プロトコル（ＨＴＴＰ）プロトコル４３５をサポートする。さらに、ストレージオペレーティングシステム４００は、ＲＡＩＤプロトコル等のディスクストレージプロトコルを実施するディスクストレージ層４４０、及び、スモール・コンピュータ・システム・インタフェース（ＳＣＳＩ）等のディスク制御プロトコルを実施するディスクドライバ層４４５を含む。

【００３６】

これらのディスクソフトウェア層とネットワークプロトコル層及びファイルシステムプロトコル層との橋渡しをするのが、ストレージオペレーティングシステム４００のファイルシステム層４５０である。一般にファイルシステム層４５０は、ディスク上で例えば４キロバイト（ＫＢ）のデータブロックを用いたブロックベースのフォーマット表現を有するファイルシステムを実施し、ｉｎｏｄｅを用いてファイルを表現する。このファイルシステムは、トランザクション要求に応じて、要求されたデータが「コア内」すなわちファイラーのメモリ３２５に存在しない場合、そのデータをボリュームからロード（取得）する処理を行なう。情報がメモリに存在しない場合、ファイルシステム層４５０は、ｉｎｏｄｅ番号を用いてｉｎｏｄｅファイル内に索引を付け、適当なエントリにアクセスしてボリュームブロック番号を取得する。次いでファイルシステム層４５０は、そのボリュームブロック番号をディスクストレージ（ＲＡＩＤ）層４４０に渡し、ＲＡＩＤ層４４０がそのボリュームブロック番号をディスクブロック番号にマッピングして、そのディスクブロック番号をディスクドライバ層４４５の適当なドライバ（例えば、ファイバーチャネルディスク相互接続上で実施されるＳＣＳＩのカプセル化）に送信する。次いでディスクドライバは、ディスクのそのディスクブロック番号にアクセスして要求されたデータをメモリ３２５にロードし、ファイラー３１０、３１２で処理する。要求が完了すると、ファイラーは、ＣＩＦＳ規格に規定されている従来の受領応答パケットを、個々のネットワーク接続３７２を介してクライアント３７０に返す。

【００３７】

ファイラーで受信されたクライアント要求についてデータストレージアクセスを実施しなければならない上記ストレージオペレーティングシステムを通るソフトウェア「パス」４７０は、代替としてハードウェアで実施することも、あるいはハードウェアとソフトウェアの組み合わせで実施することもできる。従って、本発明の代替実施形態では、ストレージアクセス要求データパス４７０が、フィールド・プログラマブル・ゲートアレイ（ＦＰＧＡ）や特定用途向け集積回路（ＡＳＩＣ）の内部に実現される論理回路として実施される場合がある。この種のハードウェア実施形態は、クライアント３７０が発行したファイルシステム要求パケット３７４に応答してファイラー３１０、３１２によって提供されるファイルサービスの性能を向上させることができる。

【００３８】

ファイルシステム層４５０の上に重なるのは、本発明の例示的形態によるスナップショット・ミラーリング（複製）・アプリケーション４９０である。以下で詳細に説明するように、（ソース側では）このアプリケーションは、スナップショットをスキャンして、スナップショットの変化をソースファイラー３１０から宛先ファイラー３１２へネットワークを介して送信する働きがある。（宛先側では）このアプリケーションは、受信した情報を基にしてミラースナップショットを更新する働きがある。従って、ソースアプリケーション及び宛先アプリケーションの特定機能は、以下で説明するように、異なる場合がある。スナップショット・ミラーリング・アプリケーション４９０は、ＴＣＰ／ＩＰ層４１５，

４１０に対する直接リンク４９２、及びファイルシステムスナップショット手段（４８０）に対する直接リンク４９４に示すように、通常の要求パス４７０の外側で動作する。このアプリケーションは、ファイルシステム層と対話してファイルの受領応答を受信するので、ファイルベースのデータ構造（具体的にはｉｎｏｄｅファイル）を用いてソースのスナップショットを宛先に複製することができる。

【００３９】

Ｃ．スナップショット手順

例示のＷＡＦＬファイルシステムに固有のＳｎａｐｓｈｏｔ（Ｒ）機能については、Ｄａｖｉｄ Ｈｉｔｚ他による「Ｆｉｌｅ Ｓｙｓｔｅｍ Ｄｅｓｉｇｎ ｆｏｒ ａｎ ＮＦＳ Ｆｉｌｅ Ｓｅｒｖｅｒ Ａｐｐｌｉａｎｃｅ」と題したＴＲ３００２に詳しい説明があり、この文献はここで参照することにより本明細書に取り込まれる。「Ｓｎａｐｓｈｏｔ」がネットワークアプライアンス社の登録商標であることに注意して欲しい。この用語は、この特許の目的で、Ｐｅｒｓｉｓｔｅｎｔ Ｃｏｎｓｉｓｔｅｎｃｙ Ｐｏｉｎｔ（ＣＰ）Ｉｍａｇｅを指すために用いられている。Ｐｅｒｓｉｓｔｅｎｔ Ｃｏｎｓｉｓｔｅｎｃｙ Ｐｏｉｎｔ Ｉｍａｇｅ（ＰＣＰＩ）とは、記憶装置（例えばディスク等）に格納されたストレージシステムをある時点で表現したものであって、特にアクティブファイルシステムをある時点で表現したものであり、そのＰＣＰＩを他の時点で作成されたＰＣＰＩから区別する名称その他一意の識別子を有する。また、ＰＣＰＩには、そのイメージが作成された時点でのアクティブファイルシステムに関するその他の情報（メタデータ）が含まれる場合もある。「ＰＣＰＩ」という用語と「スナップショット」という用語は、ネットワークアプライアンス社の商標権から外れることなく、交換可能に用いられる。

【００４０】

スナップショットは、何らかの規則的なスケジュールで作成されるのが一般的である。このスケジュールは大きな変化の影響を受ける。さらに、ファイラーに保持されるスナップショットの数も極めて不定である。格納手段が１つの場合、複数の最近のスナップショットは連続して格納し（例えば４日分のスナップショットを各々４時間の間隔で）、複数のそれよりも古いスナップショットはもっと時間間隔を広げて（例えば前の週に関する日毎のスナップショット及び前の数ヶ月についての週毎のスナップショットなど）保持する場合がある。スナップショットは、アクティブファイルシステムと共に格納され、以下で詳しく説明するように、ストレージオペレーティングシステム４００またはスナップショットミラーアプリケーション４９０から要求があったときに、ファイラーメモリのバッファキャッシュに呼び出される。しかしながら、本発明の教示の中で、様々なスナップショット作成手段及び時間調節手段を用いることができると考えられる。

【００４１】

例示的実施形態による例示的ファイルシステムｉｎｏｄｅ構造５００を図５に示す。このｉｎｏｄｅファイルのｉｎｏｄｅ、すなわちもっと普通に「ルート」ｉｎｏｄｅ５０５は、所与のファイルシステムに関するｉｎｏｄｅファイル５０８を表す情報を保持している。この例示的ファイルシステムｉｎｏｄｅ構造の場合、ルートｉｎｏｄｅ５０５は、ｉｎｏｄｅファイル間接ブロック５１０へのポインタを保持している。ｉｎｏｄｅファイル間接ブロック５１０は、１以上のｉｎｏｄｅファイル直接ブロック５１２を指しており、ｉｎｏｄｅファイル直接ブロック５１２の各々はｉｎｏｄｅファイル５０８を構成するｉｎｏｄｅ５１５への一連のポインタを保持している。図示の問題となるｉｎｏｄｅファイル５０８はｉｎｏｄｅ５１５を構成するボリュームブロックに編成され、ｉｎｏｄｅ５１５はファイルデータ（または「ディスク」）ブロック５２０Ａ、５２０Ｂ及び５２０Ｃへのポインタを保持している。この図の場合、単にｉｎｏｄｅ自体がファイルデータブロックへのポインタを保持しているように単純化して図示している。例示の実施形態では、ファイルデータブロック５２０（Ａ～Ｃ）の各々は、４キロバイト（ＫＢ）のデータを格納するようになっている。しかしながら、１つのｉｎｏｄｅ（５１５）によって所定数を超えるファイルデータブロックが参照される場合、１以上の間接ブロック５２５（仮想的に示

す）が用いられるということに注意して欲しい。それらの間接ブロックは、それらに関連するファイルデータブロックを指す（図示せず）。あるｉｎｏｄｅ（５１５）が間接ブロックを指している場合、そのｉｎｏｄｅがさらにファイルデータブロックも指すことはありえず、その逆も言える。

【００４２】

ファイルシステムは、所与のファイルシステムのスナップショットを作成する場合、図６に示すようなスナップショットｉｎｏｄｅを作成する。スナップショットｉｎｏｄｅ６０５は、実質的にファイルシステム５００のルートｉｎｏｄｅ５０５の複製である。従って例示のファイルシステム構造６００は、図５に図示したものと同じｉｎｏｄｅファイル間接ブロック５１０、ｉｎｏｄｅファイル直接ブロック５１２、ｉｎｏｄｅ５１５、及び、ファイルデータブロック５２０（Ａ～Ｃ）を有する。ユーザがファイルデータブロックを変更した場合、ファイルシステム層は、その新たなデータブロックをディスクに書き込み、新たに作成したブロックを指すようにアクティブファイルシステムを変更する。ファイル層は、その新たなデータをスナップショットに保持されるブロックには書き込まない。

【００４３】

図７は、ファイルデータブロックが変更された後の例示的ｉｎｏｄｅファイルシステム構造７００を示している。この例の場合、ディスクブロック５２０Ｃに格納されたファイルデータが変更された。例示のＷＡＦＬファイルシステムは、この変更された内容をディスク上の新たな位置である５２０Ｃ’に書き込む。新たな位置に書き込むため、ディスクブロック（５１５）に格納されたｉｎｏｄｅファイルデータは、ブロック５０２Ｃ’を指すように書き換えられる。この変更によって、ＷＡＦＬは、５１５にある変更されたデータについて新たなディスクブロック（７１５）を割り当てる。同様に、ｉｎｏｄｅファイル間接ブロック５１０をブロック７１０に書き換え、直接ブロック５１２をブロック７１２に書き換え、新たに変更されたｉｎｏｄｅ７１５を指すようにする。従って、ファイルデータブロックの変更が終った後、スナップショットｉｎｏｄｅ６０５は、元のｉｎｏｄｅファイルシステム間接ブロック５１０へのポインタを保持し、ｉｎｏｄｅ５１５へのリンクを有している。このｉｎｏｄｅ５１５は、元のファイルデータブロック５２０Ａ、５２０Ｂ、及び、５２０Ｃへのポインタを保持している。しかしながら、新たに書き込まれたｉｎｏｄｅ７１５は、未変更のファイルデータブロック５２０Ａ及び５２０Ｂへのポインタしか有していない。またｉｎｏｄｅ７１５は、アクティブファイルシステムの新たな構成を表す変更されたファイルデータブロック５２０Ｃ’へのポインタも保持している。新たなファイルシステムルートｉｎｏｄｅ７０５は、新たな構造７００を表すように確立される。何らかのスナップショットを作成したブロック（例えばブロック５１０、５１５及び５２０Ｃ）にあるメタデータは、それらがすべてのスナップショットから開放されるまで、再利用や上書きから保護されることに注意して欲しい。従って、アクティブファイルシステムルート７０５が新たなブロック７１０、７１２、７１５及び５２０Ｃ’を指していても、古いブロック５１０、５１５及び５２０Ｃは、スナップショットが完全に開放されるまで保持される。

【００４４】

本発明の例示的実施形態では、ソースは、２つのスナップショット、すなわち、宛先上の複製ファイルシステムのイメージを表す「ベース」スナップショットとソースシステムが宛先に複製しようとしているイメージである「差分」スナップショットとを用いて、宛先にミラーされたリモートスナップショットに必要な更新を実施する。一例では、ソースの立場から、差分スナップショットは最新のスナップショットを含み、ベーススナップショットはあまり最近のものではないスナップショットを含み、最新の変更の集合を宛先に提供することを可能にする。次にこの手順について、さらに詳しく説明する。

【００４５】

Ｄ．遠隔ミラーリング

スナップショットを作成するための一般的な手順の説明が終わったので、ソースファイラー３１０（図３）から遠隔の宛先ファイラー３１２へのスナップショット情報のミラーリ

ングについてさらに詳しく説明する。上で概説したように、ボリューム全体における変更されたブロックの比較に基づくスナップショットの変更差分の伝送は、ファイルシステムスナップショット全部ではなく、データの変更差分だけを転送し、小さくて高速な変更を可能にするので、有利である。しかしながら、本発明の例示的実施形態によると、宛先ミラースナップショットの遠隔差分更新について、さらに効率的及び／又は多用途な手順が考えられる。本明細書で用いられるように、「複製スナップショット」、「複製スナップショット」または「ミラースナップショット」という用語は、適当なスナップショットを保持する宛先ボリューム上のファイルシステムを大まかに指すものとして解釈すべきである（例えばスナップショットのスナップショットが含まれる）。

【００４６】

上で簡単に述べたように、この手順は、Ｑｔｒｅｅと呼ばれるボリュームの下位構成の利点を得ることができる。Ｑｔｒｅｅは、従来のＵＮＩＸ（Ｒ）ファイルシステムやＷｉｎｄｏｗｓ（Ｒ）ファイルシステムにおけるパーティションサイズによるデータの集まりに対する制限と同様の働きがあるが、ディスク上の特定領域のブロックに結び付けられていないので、制限の実質的変更について柔軟性がある。ディスクの特定の集まり（例えばｎ台のディスクからなるＲＡＩＤグループ）にマッピングされ従来のパーティションに比較的似ているボリュームとは異なり、Ｑｔｒｅｅは、ボリュームよりも高レベルで実施されるので、高い柔軟性を提供することができる。Ｑｔｒｅｅは、基本的に、ストレージオペレーティングシステムのソフトウェアにおける抽象概念である。実際、各ボリュームは複数のＱｔｒｅｅを有する場合がある。Ｑｔｒｅｅの粒度は、わずか数キロバイトの記憶容量にすることができる。Ｑｔｒｅｅ構造は、適当なファイルシステム管理者によって定義することもできるし、かかる制限を設定する適当な権限を有するユーザによって定義することもできる。

【００４７】

上で説明したＱｔｒｅｅ構成は例示的なものであり、本発明の原理はボリューム全体の方法を含めて、様々なファイルシステム構成に適用することができることに注意して欲しい。Ｑｔｒｅｅが例示の実施形態による便利な構成であることの理由の一つは、ｉｎｏｄｅファイルに識別子を利用できるからである。

【００４８】

宛先におけるソースの複製のためにデータを宛先に転送する元になる２つのソーススナップショットにおける変更を計算する処理についてさらに説明する前に、図５～図７に示すファイルブロック構造をもう一度大まかに参照しておく。ファイルのあらゆるデータブロックは、ディスクブロック（又はボリュームブロック）にマッピングされる。あらゆるディスク／ボリュームブロックは、個別のボリュームブロック番号（ＶＢＮ）を用いて一意に数えられる。各ファイルは、それらのデータブロックへのポインタを保持する１つのｉｎｏｄｅで表現される。これらのポインタはＶＢＮであり、ｉｎｏｄｅの各ポインタフィールドは内部にＶＢＮを有し、ファイルシステム（またはディスク制御）層に対する要求により適当なディスク／ボリュームブロックを読み出すことによって、ファイルのデータがアクセスされる。このディスクブロックのＶＢＮは、ｉｎｏｄｅのポインタフィールドに配置される。スナップショットは、ある時点でｉｎｏｄｅおよびそのｉｎｏｄｅの中のＶＢＮフィールドすべてを取り込む。

【００４９】

ｉｎｏｄｅ内のＶＢＮ「ポインタ」の最大数を超えてスケーリングするため、「間接ブロック」が用いられる。実質的に、ディスクブロックは、データブロックのＶＢＮが割り当てられ、それらＶＢＮで満たされ、次いでｉｎｏｄｅポインタがその間接ブロックを指す。大きなツリー構造を作ることができる複数階層の間接ブロックが存在する場合がある。間接ブロックは間接ブロック内のＶＢＮが変更されるたびに毎回、通常のデータブロックと同じ方法で変更され、その間接ブロックの変更されたデータについて新たなディスク／ボリュームブロックが割り当てられる。

【００５０】

１．ソース
図８は、ソース環境８００内部のスナップショットｉｎｏｄｅの例示的ペアを示している。例示的実施形態において、これらは、２つのスナップショットのｉｎｏｄｅファイル、すなわち、ベース８１０及び差分８１２を表している。これら２つのスナップショットは２つの時点で作成されたものであり、ベースは複製の現在のイメージを表し、差分はその複製を更新するイメージを表すものであることに注意して欲しい。２つのスナップショットの差異によって、いずれの変更を作成して遠隔の複製／ミラーに送信すべきかが決まる。各ｉｎｏｄｅファイルは、ストレージオペレーティングシステムのスナップショットマネージャ（図４の４８０）による指示に従って、ソースファイルシステムのディスク上のバッファキャッシュからソースファイルサーバメモリのバッファキャッシュにロードされる。一実施形態において、ベーススナップショット及び差分スナップショットは、ストレージオペレーティングシステムで使用されるときにストレージオペレーティングシステムにロードされる（一度に全部ではなく）。各スナップショットｉｎｏｄｅファイル８１０、８１２は、一連のストレージブロック８１４に編成される。この例の場合、ベースｉｎｏｄｅファイル８１０はボリュームブロック番号５、６、及び７で記されたストレージブロックを含み、差分スナップショットｉｎｏｄｅファイルは例えばボリュームブロック番号３、５、６、及び８のボリュームブロック番号のストレージブロックを含む。各ブロックの内部は、所定数のｉｎｏｄｅ８１６に編成される。ボリュームブロックには、それらが基礎にしている論理ファイルブロックの配置に基づいて、図示の順番で索引が付される。

【００５１】
Ｗｒｉｔｅ－ａｎｙｗｈｅｒｅファイルレイアウトの例の場合、ストレージブロックは、直ぐに上書きされたり、再利用されたりすることがない。従って、一連のボリュームブロックで構成されたファイルを変更すれば、常に新たな（新しく書き込まれた）ボリュームブロック番号が発生することになるので、その番号は古いブロックに対する適当な論理ファイルブロックオフセットとして検出することができる。ベーススナップショットｉｎｏｄｅファイルと差分スナップショットｉｎｏｄｅファイルの間の索引の所与のオフセットに変更されたボリュームブロック番号が存在することは、基礎とするｉｎｏｄｅのうちの１以上、及び、そのｉｎｏｄｅが指しているファイルに変更があったことを意味している。しかしながら、本システムはｉｎｏｄｅやポインタの変更のインジケータに他のインジケータを用いることもでき、これは固定位置ファイルシステムが実施される場合に望ましい。

【００５２】
スキャナ８２０は、索引をサーチし、ボリュームブロック番号その他の識別子を比較して、変更されたベース／差分ｉｎｏｄｅファイルスナップショットブロックを見付ける。図８の例では、ベーススナップショットｉｎｏｄｅファイル８１０のブロック４は、ファイルスキャン順番で、差分スナップショットｉｎｏｄｅファイル８１２のブロック３に対応している。これは、１以上の基礎とするｉｎｏｄｅの変更を示している。さらに、ベーススナップショットｉｎｏｄｅファイルのブロック７は、差分スナップショットｉｎｏｄｅファイルのブロック８のように見える。両ファイルにおけるブロック５及び６は、変更されていないので、何もｉｎｏｄｅその他の情報を更に処理することなく、高速にスキャンされる。従って、両スナップショットにおいて同じ索引でスキャンされたブロックは有効にバイパスして、スキャン時間を削減することができる。

【００５３】
変更があったものとして識別されたブロック対（例えばブロック７及び８）は、ｉｎｏｄｅ採集器プロセス８３０を含むソースプロセスの残りの部分に転送される。ｉｎｏｄｅ採集器は、転送されたブロックの中から（ＱｔｒｅｅＩＤに基づいて）ミラーされている選択されたＱｔｒｅｅの一部である特定のｉｎｏｄｅを識別する。この例の場合、ＱｔｒｅｅＩＤ　Ｑ２が選択されているので、ｉｎｏｄｅファイルメタデータにこの値を有するｉｎｏｄｅが「取り出され」、さらに処理される。選択されたＱｔｒｅｅの一部ではない他

のｉｎｏｄｅ（例えばＱｔｒｅｅＩＤがＱ１及びＱ３のｉｎｏｄｅ）は、採集器プロセス８３０によって破棄または無視される。複数のＱｔｒｅｅＩＤを選択し、選択されたＱｔｒｅｅ関連のうちの１つを有するｉｎｏｄｅのグループを、採集器プロセスによって抜き出すこともできる。

【００５４】

次いで、変更されたブロックから適切に「取り出された」ｉｎｏｄｅは、変更されたｉｎｏｄｅ８４２の実行中リストすなわちキュー８４０に入れられる。これらのｉｎｏｄｅは、図示のように不連続の番号で記される。キュー８４０内の各ｉｎｏｄｅは、ｉｎｏｄｅハンドラまたはワーカー８５０、８５２及び８５４に渡され、それらのｉｎｏｄｅワーカーが利用できるようになる。図８Ａは、ｉｎｏｄｅ採集器プロセス８３０が与えられたｉｎｏｄｅをワーカーで処理するためにキューに渡すか否かを判定するのに使用するルールの基本集合の詳細を示す表８３５である。

【００５５】

ｉｎｏｄｅ採集器プロセス８３０は、（１）対象ｉｎｏｄｅ（与えられたｉｎｏｄｅ番号）のベーススナップショットのバージョンが選択されたＱｔｒｅｅに割り当て及び配置される（ボックス８６０）のか、（２）対象ｉｎｏｄｅの差分スナップショットのバージョンが選択されたＱｔｒｅｅに割り当て及び配置される（ボックス８６２）のかを問い合わせる。ベースバージョン及び差分バージョンがいずれも選択されたＱｔｒｅｅに割り当て及び配置されない場合、両ｉｎｏｄｅを無視し（ボックス８６４）、ｉｎｏｄｅバージョンの次のペアを参照する。

【００５６】

ベースｉｎｏｄｅは選択されたＱｔｒｅｅに割り当て及び配置されないが、差分ｉｎｏｄｅは選択されたＱｔｒｅｅに割り当て及び配置される場合、これは差分ファイルが追加されたことを意味しているので、適当なｉｎｏｄｅ変更をワーカーに送信する（ボックス８６６）。同様に、ベースｉｎｏｄｅは選択されたＱｔｒｅｅに割り当て及び配置されるが、差分ｉｎｏｄｅは選択されたＱｔｒｅｅに割り当て及び配置されない場合、これはベースファイルが消去されたことを意味しているので、これがデータストリームの形で宛先に送信される（ボックス８６８）。

【００５７】

最後に、ベースｉｎｏｄｅ及び差分ｉｎｏｄｅが両方とも選択されたＱｔｒｅｅに割り当て及び配置される場合、プロセスは、ベースｉｎｏｄｅと差分ｉｎｏｄｅが同じファイルを表すものであるか否かを問い合わせる（ボックス８７０）。それらが同じファイルを表している場合、そのファイルまたはそのメタデータ（パーミッション、所有者、パーミッションなど）は変更された可能性がある。これは、採集器プロセスによって検査されている様々なバージョンのｉｎｏｄｅ番号に対する異なる生成番号で示される。この場合、変更されたファイルが送信され、以下で説明するように、ｉｎｏｄｅは、正確な変更を判定するためにバージョンの比較を行なうように働く（ボックス８７２）。ベース及び差分がまったく同じファイルではない場合、これは、ベースファイルの消去及び差分ファイルの追加を意味している（ボックス８７４）。そのような場合、採集器によって差分ファイルの追加がワーカーキューに記入される。

【００５８】

図８Ｂは、ベーススナップショット８１０における変更されたブロックの例（ブロック１０）、及び、差分スナップショット８１２における変更されたブロック（ブロック１２）の例の各々に含まれる情報の詳細を示している。ｉｎｏｄｅ２８００は、ベースｉｎｏｄｅファイルにも差分ｉｎｏｄｅファイルにも割り当てられていない。これは、そのファイルがファイルシステムに追加されたことを意味している。また、ｉｎｏｄｅ採集器プロセスは、このｉｎｏｄｅが適当なＱｔｒｅｅ　Ｑ２（この例の場合）に配置されていることも記している。このｉｎｏｄｅは、ファイル全体が新しいものであるという注釈とともに、変更されたｉｎｏｄｅキューに送信され、処理される。

【００５９】

ｉｎｏｄｅ２８０１は、両方のｉｎｏｄｅファイルに割り当てられている。このｉｎｏｄｅは、適当なＱｔｒｅｅ　Ｑ２に配置され、２バージョンのこのｉｎｏｄｅが同じ生成番号を共有している。この時点ではファイルデータ自体に変更があったか否かが分からないので、ｉｎｏｄｅ採集器はこのペアを変更されたｉｎｏｄｅキューに送信し、ワーカーがいずれのデータに変更があったかを判定する。ｉｎｏｄｅ２８０２は、ベースｉｎｏｄｅファイルに割り当てられているが、差分ｉｎｏｄｅファイルには割り当てられていない。このｉｎｏｄｅのベースバージョンは、適当なＱｔｒｅｅ　Ｑ２にあった。これは、このｉｎｏｄｅが消去されたことを意味している。ｉｎｏｄｅ採集器は、同様にこの情報をワーカーに送信する。最後に、ｉｎｏｄｅ２８０３は、ベースｉｎｏｄｅファイルに割り当てられていて、差分ｉｎｏｄｅファイルにも再割り当てされている。ｉｎｏｄｅ採集器８３０は、２つのバージョン間で（＃１と＃２の間で）生成番号が変更されたからであることから、これを判断することができる。このｉｎｏｄｅが表すファイルはｉｎｏｄｅ２８００と同様にＱｔｒｅｅに追加されているので、このｉｎｏｄｅは、ファイル全体が新しいものであるという注釈とともに、変更されたｉｎｏｄｅキューに送信され、処理される。

【００６０】

所与の時点で、所定数のワーカーがキュー８４０に対して処理を行なう。例示の実施形態の場合、ワーカーの機能は、キュー内のｉｎｏｄｅグループに対して並行で行なわれる。つまり、ワーカーは、特に順番なくいったんキューからｉｎｏｄｅを取得すると、それらが利用可能になるや否や、さらなるｉｎｏｄｅをキューから自由に処理することができる。スキャン８２０及び採集器８３０など、他のプロセスも、順番全体の中でインタリーブされる。

【００６１】

ワーカーの機能は、ファイル及びディレクトリの各スナップショットのバージョン間の変化を判定することである。上記のように、ソーススナップショットミラーアプリケーションは、２つのスナップショットにおけるｉｎｏｄｅの２つのバージョンを分析して、それらのｉｎｏｄｅのポインタを比較するようになっている。それらのポインタの２つのバージョンが同じブロックを指している場合、そのブロックは変更されなかったことが分かる。また、間接ブロックへのポインタが変更されなかった場合、その間接ブロックにデータの変更はないので、間接ブロックのポインタが変更されることもありえず、従って、ツリーの下方にあるデータブロックに変更はなかったことになる。これが意味するのは、２つのスナップショット間でほとんど変更されていない巨大なファイルの場合、そのツリー内の各データブロックに対するＶＢＮ「ポインタ」を読み飛ばすことにより、それらのデータブロックのＶＢＮに変更があったか否かの確認をスキップすることができるということである。

【００６２】

例として、ワーカー８５０の処理を図９に示す。ワーカー８５０は、変更のあったｉｎｏｄｅ対を受信すると、各ｉｎｏｄｅ（ベース及び差分の各々）９１０及び９１２をスキャンして、各々のブロック間でファイルオフセットが一致するか否かを判定する。この例の場合、ブロック６及び７が一致しない。次いで、ブロック６及び７各々の下方へスキャンを続け、基礎とする間接ブロック８／９（９２０）及び８／１０（９２２）に到達する。ここでも、ファイルオフセット比較は、ブロック８が両方とも共通ブロック９３０に到達したことを示す（従って変更されていない）。逆に、ブロック９及び１０は、オフセットが異なるため一致せず、変更されたブロック９４０及び９４２を指している。変更されたブロック９４２及び上記メタデータは、宛先の複製スナップショット（以下で説明する；図８も参照）に伝送するため、単独で取り出すことができる。例示の実施形態のツリーは４階層の深さまであるが、この手順は、いかなる階層数にも適用することができる。また、ツリーは複数の変更されたブランチを含む場合があり、ワーカーは、それらの変更がすべて識別されるまで、それらのブランチの各々を再帰的にたどらなければならない場合もある。従って、各ｉｎｏｄｅワーカーは、以下に説明する方法で伝送するために、それら

の変更をネットワークに提供する。特に、古い、消去されたブロックに関する新たなブロック及び情報が宛先に送信される。同様に、変更されたブロックに関する情報も送信される。

【００６３】

注意して欲しいのは、この例ではほとんどあらゆるデータ構造がファイルになっているので、上記のプロセスは、ファイルデータだけでなく、ディレクトリ、アクセスコントロールリスト（ＡＣＬ）、及び、ｉｎｏｄｅファイル自体にも適用することができるということである。

【００６４】

さらに注意して欲しいのは、このソース手順は、ボリュームｉｎｏｄｅファイル全体を含む、あらゆる粒度のファイルシステム構成に適用することができるということである。本来のＱｔｒｅｅ構成を用いることにより、そのボリュームの既知のサブセットを複製するための迅速で効率的な方法が得られる。

【００６５】

２．ソースと宛先の間の通信

図１０を参照すると、ソーススナップショットから複製された宛先スナップショットへの変更の伝送が、概観１０００に図示されている。既に説明したように、古いスナップショット及び新たなスナップショットは、対象ボリュームのＱｔｒｅｅその他の選択された下位構成に対応する変更されたｉｎｏｄｅを、ｉｎｏｄｅ採集器８３０に提示する。これらの変更されたｉｎｏｄｅはキュー８４０に配置され、次いで、ｉｎｏｄｅワーカー８５０、８５２及び８５４の集合が、各々のツリーの変更を調べる。ｉｎｏｄｅワーカーは各々、変更情報を含むメッセージ１００２、１００４及び１００６をソースパイプライン１０１０に送信する。このパイプラインはファイルシステムデータを１つのデータストリームにパッケージングしてネットワーク層に送信するための手段を実現する方法の一例にすぎないことに注意して欲しい。これらのメッセージは先ず受信器１０１２にルーティングされ、受信器がこれらのメッセージを収集し、ネットワークを介して送信すべきスナップショット変更情報を含むグループとしてそれらのメッセージをアセンブラ１０１４に送信する。ここでも、本明細書に記載する「ネットワーク」は、ソースと宛先が同じファイルサーバ、ボリューム上にある場合でも、ソース下位構成から宛先下位構成へのボリューム下位構成（例えばＱｔｒｅｅ）変更データの伝達を容易にするいかなるものも含むように広く解釈しなければならず、実際、（「ＳＹＳＴＥＭ ＡＮＤ ＭＥＴＨＯＤ ＦＯＲ ＲＥＭＯＴＥ ＡＳＹＮＣＨＲＯＮＯＵＳ ＭＩＲＲＯＲＩＮＧ ＵＳＩＮＧ ＳＮＡＰＳＨＯＴＳ」と題する、上で取り込まれた米国特許出願に記載されている巻き戻しの場合では、）異なる時点で同じ下位構成になっている。同じボリュームに戻る経路として用いられる「ネットワーク」の一例は、ループバックである。アセンブラ１０１４は、ネットワーク３１８を介して情報のデータストリームを送信するため、宛先が予測及び理解することのできる特別なフォーマット１０２０を作成する。ネットワーカー１０１６は、その組み立てられたデータストリームを受信して、それをネットワーク層に転送する。このフォーマットは、ＴＣＰ／ＩＰ等、信頼性の高いプロトコルの中にカプセル化されるのが普通である。カプセル化はネットワーク層で実施することができ、この層が例えばフォーマットされた複製データストリームのＴＣＰ／ＩＰパケットを作成する。

【００６６】

フォーマット１０２０について、以下でさらに説明する。概して、このフォーマットの使用は、複数のプロトコル属性（例えば、ＵＮＩＸ（Ｒ）のパーミッション、ＮＴのアクセスコントロールリスト（ＡＣＬ）、複数のファイル名、ＮＴストリーム、ファイルタイプ、ファイル作成／変更など）を有することによって予測される。また、このフォーマットは、ストリーム内のデータ（すなわち、ファイル中の特定データのオフセット位置や、ファイルがファイルオフセット中に空の状態を保つ必要がある「ホール」を有するか否か）を識別ものでなければならない。また、ファイルの名称もこのフォーマットで中継しなければならない。さらに一般的に言うと、このフォーマットは更に、基礎とするネットワー

クプロトコルや装置（テープやローカルディスク／不揮発性記憶装置の場合）プロトコル及びファイルシステムとは無関係なものにする必要があり、つまり、その情報をシステム「不認識」なものにし、特定のオペレーティングシステムに束縛されないものにすることによって、異なるベンダーのソースシステムと宛先システムがその情報を供給できるようにする必要がある。従って、このフォーマットは、データストリームの外部に何も情報を必要としない自己表現的なものにする必要がある。このような方法で、第１のタイプのソースファイルディレクトリを別のタイプの宛先ファイルディレクトリに直ちに変換することができる。また、ソースオペレーティングシステムまたは宛先オペレーティングシステムに対する新たな改良が古いバージョンの互換性に悪影響を与えないようにする際に、拡張性も可能になる。具体的には、オペレーティングシステムによって認識されないデータ集合（例えば新しいヘッダ）は、必ず無視されるようにするか、あるいは、システムクラッシュその他の望ましくない故障をトリガすることなく予測的方法で処理されるように（すなわち、ストリームが下位互換になる）しなければならない。また、このフォーマットは、ファイルシステム全体の表現も、何らかのファイルまたはディレクトリ内部の変更されたブロック／情報だけの表現も、伝達できるものにしなければならない。さらに、このフォーマットは、ネットワーク及びプロセッサのオーバヘッドを最小限にしなければならない。

【００６７】

変更された情報は、ネットワークを介して転送され、宛先パイプライン部分１０３０で受信される。このパイプラインも、ネットワークからＴＣＰ／ＩＰパケットをＴＣＰ／ＩＰにカプセル化されたスナップショット複製データストリームフォーマット１０２０に読み出すためのネットワーカー１０３２を含む。データ読み取り及びヘッダ分離器１０３４は、様々なフォーマットヘッダ（以下で説明する）に格納された情報に作用することで、入ってきたフォーマット１０２０を認識して応答する。ファイル書き込み器１０３６は、そのフォーマットから取り出したファイルデータを宛先ファイルシステム内の適当な位置に配置する役割がある。

【００６８】

宛先パイプライン１０３０は、データ及びディレクトリ情報を、以下で詳しく説明するメイン宛先スナップショットミラープロセス１０４０に転送する。宛先スナップショットミラープロセス１０４０は、受信したスナップショット変更に基づいて宛先側に新たな複製ファイルシステムディレクトリ階層を構築するディレクトリステージ１０４２から構成される。簡単に説明すると、このディレクトリステージは、受信したフォーマットされた情報に基づいて、ファイルを消去または移動するものである。宛先からソースへのｉｎｏｄｅのマップが生成され、更新される。この方法で、ソースファイルシステムのｉｎｏｄｅ番号は、宛先ファイルシステムの対応する（しかしながら通常は異なる）ｉｎｏｄｅ番号に関連付けられる。変更または消去されたディレクトリエントリ１０５２がディレクトリステージ内の適当なディレクトリ再構築ステージで再利用されるか、または複製スナップショットから除去されるまでそれらエントリを維持するため、テンポラリすなわち「一時的」ディレクトリ１０５０が作成されることに注意して欲しい。さらに、宛先ミラープロセスのファイルステージ１０４４は、ディレクトリステージで作成したファイルを、関連ファイルヘッダから分離した情報に基づいてデータに格納する。

【００６９】

ソーススナップショット変更を編成するフォーマットを、図１１及び図１２に概略的に示す。例示の実施形態の場合、このフォーマットは、約４ＫＢのブロックに編成される。しかしながら、ヘッダのサイズ及び構成は、実施形態によって様々に異なるものでよい。特定の「ヘッダタイプ」によって識別される４ＫＢヘッダ（図１１の符号１１００）が存在する。ほぼ2メガバイト（２ＭＢ）毎の変更データについて、基本データストリームヘッダ（「データ」）が設けられる。図１１を参照すると、４ＫＢの単体ヘッダには、３つの部分、すなわち１ＫＢの汎用部分、２ＫＢの非汎用部分、及び、１ＫＢの拡張部分が含まれる。拡張部分は使用されないが、最近のバージョンでは利用することができる。

【００７０】
汎用部分１１０２には、ヘッダタイプ１１１０の識別子が含まれる。ヘッダタイプ単体（すなわち、後に関連データを伴わないヘッダ）は、データストリームの開始、データストリームの一部の終端、データストリームの終端、ヘッダにカプセル化された消去されたファイルのリスト、または、ＮＴ　ｓｔｒｅａｍｄｉｒの関係を示すことができる。最近のバージョンのＷｉｎｄｏｗｓ（Ｒ）　ＮＴは、複数のＮＴ「ストリーム」を特定のファイルネームに関連付けることができる。ストリームに関する説明は、Ｋａｙｕｒｉ　Ｐａｔｅｌ他による「ＳＹＳＴＥＭ　ＡＮＤ　ＭＥＴＨＯＤ　ＦＯＲ　ＲＥＰＲＥＳＥＮＴＩＮＧ　ＮＡＭＥＤ　ＤＡＴＡ　ＳＴＲＥＡＭＳ　ＷＩＴＨＩＮ　ＡＮ　ＯＮ－ＤＩＳＫ　ＳＴＲＵＣＴＵＲＥ　ＯＦ　Ａ　ＦＩＬＥ　ＳＹＳＴＥＭ」と題した米国特許出願第０９／８９１１９５号に記載があり、その教示はここで参照することにより広く本発明に取り込まれる。また、汎用部分１１０２には、ヘッダが壊れていないことを確認するためのチェックサム１１１２も含まれる。さらに、ソース及び宛先が複製の進行を追跡するために用いる「チェックポイント」１１１４等、その他のデータも設けられる。ヘッダタイプのリストを設けることにより、宛先は比較的容易に下位互換モードで動作することができ、つまり、宛先バージョンの制限内で認識できるヘッダは普通に処理する一方、宛先が理解できないヘッダタイプ（比較的新しいバージョンのソースから供給される）はもっと簡単に無視することができる。
【００７１】
ヘッダ１１００の非汎用部分１１０４のデータの種類は、ヘッダタイプに応じて決まる。基本的なヘッダの場合、その非汎用部分には、後続のデータ送信に使用するファイルオフセット（１１２０）、消去ファイル（ソース側でもう使用しなくなったファイルやその生成番号の変更をリストする単体ヘッダ）、または、その他ヘッダ固有の情報（１１２４、以下で説明する）に関する情報が含まれる。この場合も、データストリームフォーマット内の適当な位置に様々な単体ヘッダが挿入される。各ヘッダは、データ集合（消去されたファイル等）または後続の情報（ファイルデータ等）を参照するように構成される。
【００７２】
図１２は、例示の複製データストリームのフォーマット１０２０をさらに詳細に表したものである。複製データストリームのフォーマットは、「データストリームの開始」という種類の単体データストリームヘッダ１２０２で始まる。このヘッダには、データストリームの属性を記述するソースによって生成された非汎用部分１１０４のデータが格納される。
【００７３】
フォーマット１０２０において、次の一連のヘッダは、様々な「パート１」情報（１２０４）を定義する。重要なのは、送信される各ディレクトリデータ集合の前に、非汎用データを有しない基本ヘッダがくることである。変更されたディレクトリだけが送信され、それらは特定の順番で到着する必要がない。また、何らかの特定ディレクトリからのデータが連続している必要もない。各ディレクトリエントリは、４ＫＢブロックにロードされる。あふれたものは、いずれも新たな４ＫＢブロックにロードされる。各ディレクトリエントリは、後ろに１以上の名称が続くヘッダである。各ディレクトリエントリは、続いてｉｎｏｄｅ及びディレクトリ名を記述する。ＮＴストリームディレクトリも送信される。
【００７４】
パート１フォーマット情報１２０４は、関連ＡＣＬを有するあらゆるファイルについてＡＣＬ情報も提供する。ＡＣＬの関連ファイルデータより前にＡＣＬを送信することにより、宛先はファイルデータが書き込まれる前にＡＣＬを設定することができるようになる。ＡＣＬは、「通常の」ファイルフォーマットで送信される。消去ファイル情報（上記の）は、こうした情報と共に１以上の単体ヘッダの非汎用部分１１０４（もしあれば）に含められて送信される。この情報を前もって送信することにより、ディレクトリ構築機は、移動と消去を区別することができるようになる。
【００７５】

パート１フォーマット情報１２０４は、ＮＴストリームディレクトリ（ｓｔｒｅａｍｄｉｒ）関係情報も保持する。１以上の単体ヘッダ（もしあれば）は、ＮＴストリームを含む宛先ファイルサーバのあらゆる変更されたファイルまたはディレクトリを、それらのストリームに変更があったか否かに関わらず通知する。この情報は、ヘッダ１１００（図１１）の非汎用部分１１０４に含められる。

【００７６】

パート１フォーマット情報１２０４は、複製データストリームにおける、シンボリックリンク、名前付きパイプ、ソケット、ブロックデバイス、または、キャラクタデバイスのあらゆる変更について、特別なファイルを含む。それらのファイルは、複製ファイルシステムにファイルを格納する前にその複製ファイルシステムを作成するためのインフラストラクチャを構築する際に、宛先を補助するのに必要となるため、最初に送信される。特別なファイルは、ＡＣＬに似たものであり、通常ファイルの形態で送信される。

【００７７】

様々なパート１情報１２０４を送信した後、このフォーマットは、「データストリームのパート１の終端」ヘッダ１２０６を必要とする。これは、非汎用部分１１０４にデータを有しない基本ヘッダである。このヘッダは、パート１が終ったので今度はファイルデータを期待するということを宛先に告げるものである。

【００７８】

パート１情報の後、このフォーマットは、ファイル及びストリームデータ１２０８を与える。ファイル内のあらゆる２ＭＢ以下の変更されたデータについて、基本ヘッダ１２１０が設けられ、その後にファイルデータ１２１２自体が続く。これらのファイルは、特定の順番で書き込む必要のないデータ、または、連続している必要のないデータで構成される。さらに、図１１のヘッダを参照すると、この基本ヘッダには、汎用部分１１０２内部の（この例の場合）「ホールアレイ」１１３２と協働する、非汎用部分１１０４に関連するブロック番号データ構造１１３０が含まれる。ホールアレイは空き空間を意味している。この構造は、実質的に、ホールアレイからファイル内の対応するブロックへのマッピングを提供する。この構造は、データブロックまたはホールを書き込むべき場所を宛先に指示する。

【００７９】

一般に、ファイル（１２１２）は、４ＫＢの塊で書き込まれ、最大でも５１２個の塊（２ＭＢ）ごとに基本ヘッダを有する。同様にストリーム（これも１２１２）も、４ＫＢ塊の通常ファイルのように送信され、ヘッダ間が最大でも２ＭＢになっている。

【００８０】

最後に、複製データストリームフォーマット１０２０の終端には、「データストリームの終端」というタイプの単独ヘッダからなるフッタ１２２０で印が付けられる。このヘッダは、非汎用部分１１０４（図１１）に何もデータを持たない。

【００８１】

３．宛先

遠隔の宛先（例えば、遠隔のファイルサーバ、遠隔のボリューム、遠隔のＱｔｒｅｅ、又は同様のＱｔｒｅｅなど）は、ネットワークを介してソースファイルサーバからフォーマットされた情報を受信すると、新たなＱｔｒｅｅを作成するか、または、既存のミラーＱｔｒｅｅ（または、他の適当な組織的構造）を変更し、それをデータで埋める。図１３は、宛先スナップショットミラープロセス１０４０の詳細を示すものである。上で簡単に説明したように、このプロセスは、２つの主要部分、すなわちディレクトリステージ１０４２とデータ又はファイルステージ１０４４から構成される。

【００８２】

ディレクトリステージ１０４２は、ソースからデータストリームを送信する際に、最初に呼び出される。このステージは、複数の個別部分から構成される。これらの部分は、すべてパート１フォーマット（非ファイル）のデータを取り扱うように設計される。例示的実施形態において、パート１のデータは、宛先に読み込まれ、ファイルとしてローカルに格

納され、次いでローカル記憶装置から処理される。しかしながら、このデータはリアルタイムに到着するように処理することも可能である。

【００８３】

具体的には、ディレクトリステージ１０４２の最初の部分は、消去されたファイルヘッダの処理（１３１０）を含む。消去されたファイルについては、ｉｎｏｄｅマップ内のエントリが消去され、複製された宛先スナップ上のマッピングされたｉｎｏｄｅとソーススナップ上のｉｎｏｄｅとの関連が切断される。

【００８４】

次に、ディレクトリステージは、ツリークリーニング処理（１３１２）を請け負う。このステップは、複製されたスナップショットディレクトリ１３３０から、ソーススナップショット上で変更のあったディレクトリエントリをすべて除去する。データストリームフォーマット（１０２０）は、いずれのディレクトリエントリが追加または消去されたかを示す。実際、このフォーマットには、ベースバージョンのディレクトリからのディレクトリエントリと、差分バージョンのディレクトリからのディレクトリエントリとの両方が存在する。宛先スナップショットミラーアプリケーションは、このフォーマットされたデータストリームを宛先ディレクトリフォーマットに変換し、各エントリがｉｎｏｄｅ番号、関連名称のリスト（例えば、様々なマルチプロトコルネーム）及び、「作成」又は「消去」値を含むようにする。一般に、各ファイルは、そのファイルに関連する生成番号を有する。タプルを形成するｉｎｏｄｅ番号及び生成番号は、ファイルシステム内部でディレクトリがファイルにアクセスするために用いられる。ソースはこのタプル情報を前記フォーマットで宛先に送信し、適当なタプルが宛先システムに格納される。既存の宛先ファイルに関して、古くなった生成番号は、そのファイルがソースから消去されたことを示している。生成番号の使用については、以下でさらに説明する。

【００８５】

宛先は、ベースディレクトリエントリを消去として処理し、差分ディレクトリエントリを追加として処理する。移動またはリネームされたファイルは、消去（古いディレクトリからの、または古い名称の）として処理された後、追加（新たなディレクトリに対する、または新たな名称の）として処理される。消去その他の変更がなされたディレクトリ１０５２は、テンポラリすなわち「一時的」ディレクトリに移動され、ユーザがその場所にアクセスできないようにされる。変更されたエントリはアクティブファイルシステムの対象から完全に除去してしまうのではなく、一時的ディレクトリによって側に移動しておくことが可能になる。一時的ディレクトリエントリ自体がデータを指しているので、データが消去されたり、ディレクトリへのリンクが失われたりすることを、完全に防止することができる。

【００８６】

宛先に対するＱｔｒｅｅの基礎転送の際には、データストリーム内のファイル及びディレクトリすべてについての幅優先探索として、Ｑｔｒｅｅのルートからディレクトリステージツリー構築プロセスが実施される。次いでディレクトリステージは、ツリー構築プロセスを実施し、それらのファイルについてのスタブエントリを用いてすべてのディレクトリを構築する。しかしながら、図示の差分ディレクトリステージ（１０４２）は、本明細書に典型的に記載されるように、ツリー構築プロセス（１３１４）の基礎転送とは異なり、ソースと宛先との両方に現在存在するすべての変更されたディレクトリ（すなわち、転送より前に存在していた変更されたディレクトリ）を含むディレクトリキューから開始される。次いで差分ディレクトリステージツリー構築プロセスは、上で幅優先方法と呼んだ方法に従って、残りのディレクトリを処理する。

【００８７】

効率を高めるため、ソース側は、パス名ではなく、ｉｎｏｄｅ番号及びディレクトリブロックに依存している。通常、宛先上の複製ディレクトリツリー（この例の場合、Ｑｔｒｅｅ）のファイルは、ソース上で対応するファイルがすでに使用されているｉｎｏｄｅ番号の受信を待機することができない（可能であっても）。そのため、宛先にはｉｎｏｄｅマ

ップが作成される。図１４に全体を示すこのマップ１４００によって、ソースは、ソース上の各ファイルを宛先に関連付けることが可能になる。このマッピングは通常、ファイルオフセットに基づく。例えば、「ｉｎｏｄｅ８７７にオフセット２０ＫＢ」を有する受信されたソースブロックは、宛先ｉｎｏｄｅ９９１２に複製される。次いでこのブロックは、宛先ファイルの適当なオフセットに書き込むことができる。

【００８８】

詳しくは、ｉｎｏｄｅマップ１４００の各エントリは、ソーススナップショット上の各ｉｎｏｄｅについてエントリを有する。マップの各ｉｎｏｄｅエントリ１４０２には、ソースｉｎｏｄｅ番号（１４０４）が索引付けされていて、このソースｉｎｏｄｅ番号を介してアクセスすることができる。これらのソースｉｎｏｄｅは、マッピングされる宛先ｉｎｏｄｅの順番とは無関係に、連続的かつ単調増加する順番でマップにリストされる。このマップは、各ソースｉｎｏｄｅ番号（１４０４）の下に、マッピングされたｉｎｏｄｅがソース上の現在のファイルと一致することを検証するためのソース生成番号（１４０６）、宛先ｉｎｏｄｅ番号（１４０８）、及び、宛先生成番号（１４１０）を含む。上記のように、ｉｎｏｄｅ番号と生成番号は、合わせて対応するファイルシステムの関連ファイルに直接アクセスするのに必要なタプルを構成する。

【００８９】

格納される宛先に関してソース生成番号が上向きにインクリメントされるので、ソース生成番号を維持することにより、宛先は、ソース上でファイルの変更や消去があったか否か（及び、そのソースに関する再割り当てされたｉｎｏｄｅ）を判定することができる。ｉｎｏｄｅの変更があったことをソースが宛先に通知する場合、ソースはタプルを宛先に送信する。このタプルは、ソースシステム上のｉｎｏｄｅを一意に識別する。まったく新しいファイルやディレクトリを作成（例えば「作成」）しなければならないことをソースが宛先に示すたびに、宛先ファイルシステムはそのファイルを作成する。宛先は、そのファイルを作成すると、自分のｉｎｏｄｅマップ１４００の新たなエントリにデータを登録する。既存のファイルやディレクトリを消去しなければならないことをソースが宛先に示すたびに、宛先は、そのファイルを消去し、ｉｎｏｄｅマップのエントリをクリアする。注意して欲しいのは、ファイルを変更する場合、ソースは、使用されるタプル及びデータを送信するだけでよいということである。宛先は、ｉｎｏｄｅマップからそのソースｉｎｏｄｅのエントリをロードする。ソース生成番号が一致した場合、そのファイルは宛先上に既に存在しているので変更しなければならないということが分かる。宛先は、ｉｎｏｄｅマップに記録されたタプルを用いて宛先ｉｎｏｄｅをロードする。最後に、宛先は、そのｉｎｏｄｅを用いることにより、ファイル変更を適用することができる。

【００９０】

ツリー構築プロセスの一部として、再利用されるエントリは、一時的ディレクトリから複製スナップショットディレクトリ１３３０に「移動」して戻される。従来、ファイルの移動は、移動するファイルの名前と移動される先のファイルの名前とを知っている必要があった。一時的ディレクトリでは、この移動されるファイルの元の名前を簡単に得ることができない。また、完全な移動には２つのディレクトリ（一時ディレクトリ及び複製スナップショット）を変更しなければならず、さらなるオーバヘッドを伴う場合がある。

【００９１】

しかしながら、例示の実施形態では、宛先で受信されたソースｉｎｏｄｅがｉｎｏｄｅマップ１４００のｉｎｏｄｅを指していれば、ディレクトリステージが所望のファイル名を有するファイルエントリを作成する（現在の構築スナップショットディレクトリ１３３０に）。この名称は、ソースで作成された名称と同じでよい。スナップショットディレクトリ１３３０上のそのファイルと一時ディレクトリのエントリとの間に、ハードリンク１３３２を作成する（例えば、ＵＮＩＸ（Ｒ）ベースのリンクは、１つのファイルに複数の名前を割り当てることができる）。エントリをそのようにリンクすることによって、そのエントリは、一時的ディレクトリとスナップショットディレクトリ自体のファイルとの両方によって指し示されることになる。データストリーム転送の最後に一時的ディレクトリの

ルートが最終的に消去される場合（一時的を削除することにより）、そのハードリンクはそのエントリに対して維持され、一時ディレクトリ内の特定エントリが消去されたり再利用されたりしないことを保証し（そのエントリのリンク数が１以上であると仮定した場合）、新たなディレクトリ上のそのファイルからデータへのパスが維持される。新しく構築したツリーに最終的に関連付けられることになるあらゆる一時エントリが同時にハードリンクされ、これによって一時ディレクトリの消去に対処する。逆に、再リンクされなかった一時エントリは残らず、事実上一時ディレクトリが消去されたときに永久に消去される。

【００９２】

マッピング及び生成番号の使用によって、ソースからのデータストリームにおける高価な（処理上の観点から）従来の完全なパス名（または相対パス名）の使用を回避できることが、分かったはずである。ソース上で変更されたファイルは、ソースや宛先にディレクトリをロードすることなく、宛先上で更新することができる。これによって、ソースから必要となる情報、及び、必要な処理量が制限される。また、ソースはディレクトリ処理のログを維持する必要もない。同様に、宛先は、現在のファイルシステム状態の集中的知識を維持する必要がないので、複数のサブディレクトリを同時に処理することができる。最後に、ソース及び宛先はいずれも、自動的に消去されたファイル等、消去されたファイルを明示的に追跡する必要がない。むしろ、ソースが消去されたファイルのリストを送信するだけで、宛先はこのリストを用いてｉｎｏｄｅマップを確認することができる。このように、ファイルを消去するためにツリーを複数回にわたって選択的に検査する必要はなく、転送の終了時に一時ディレクトリを単に除去することが、唯一のファイルクリーニングステップである。

【００９３】

ディレクトリステージ１０４２は、ツリー構築中（サブステップ１３１６）にディレクトリを処理する際に、ディレクトリ上に何らかのＡＣＬを設定する。上記のように、ファイルに対するＡＣＬ及びＮＴストリームの関係は、適当な単体ヘッダに格納される。その後、以下で説明するファイルステージで、それらのＡＣＬがファイルに設定される。ＮＴストリームは、ファイル自体が作成されるときに、そのファイルに作成される。ＮＴストリームは、実際にはディレクトリであるため、そのエントリはディレクトリフェイズの一部で処理される。

【００９４】

新たなディレクトリツリーは、データを有しないファイルや、古いデータを有するファイルを保持する場合がある。「パート１の終端」フォーマットヘッダを読み出すと、宛先ミラープロセス１０４０は、ファイルステージ１０４４に入り、そこでディレクトリツリーによって参照されるスナップショットデータファイル１３４０をデータ（例えば変更データ等）で埋める。図１５は、ソースから受信したファイルデータ１５０２を書き込むための単純化した手順１５００を示している。概して、４ＫＢブロックに入れられた各（最大）２ＭＢのデータは、対応するｉｎｏｄｅ番号に到着する。対応するエントリ１４０２は、ｉｎｏｄｅマップ１４００から調べられる。そのデータから適当なオフセット１５０４が導出され、それが所定の空の宛先スナップショットファイル１３４０に書き込まれる。

【００９５】

ディレクトリステージ１０４２及びデータステージ１０４４の終わりで、すべてのディレクトリ及びファイルが処理されると、ソースからのデータストリーム転送は完了し、新たに複製されたスナップショットが自動的にユーザにエキスポートされる。この時点で、一時ディレクトリ１０５０（まだ再構築ツリーに「移動」して戻されていない何らかのエントリを含む）の内容が消去される。

【００９６】

宛先上での複製スナップショットの最初の作成（「レベル０」転送）の後、上記の一般的手順が続く。レベル０転送と通常更新との違いはベーススナップショットが存在しないことであり、そのためこの比較は、差分スナップショットの情報を、変更ではなく常に追加

及び作成として処理する。宛先ミラーアプリケーションは、既に知っている何らかのディレクトリを処理することにより、ツリー構築を開始する。宛先で作成される最初のディレクトリは、単に複製スナップショットのルートディレクトリ（Ｑｔｒｅｅルート）である。宛先ルートはｉｎｏｄｅマップ上に存在する。ソースは最終的にルート（ルートが到着するまで受信した他のファイルはバッファリングすることができる）を送信し、そのルートが既存の宛先ルートにマッピングされる。次いで、ルートで参照するファイルが宛先で受信及び読み出しされるにつれて、「作成」プロセスで、それらがマッピングされる。最終的に、ディレクトリ全体が作成され、次いでデータファイルが格納される。この後、複製ファイルシステムが完了する。

【００９７】

Ｅ．巻き戻し（ロールバック）

上記のように、ソース及び宛先は一般に、異なる時点で同じＱｔｒｅｅになる可能性がある。この場合、「巻き戻し」手順を用いることにより、スナップショットに対する変更差分が未完成になる可能性があると考えられる。本質的に、図８を参照する上記のベーススナップショット更新プロセス及び差分スナップショット更新プロセスは、災害から復旧させ、アクティブファイルシステムを所与のスナップショットの状態に戻すように、逆方向に実施される。

【００９８】

図１６を参照すると、この図は、例示的実施形態による一般的な巻き戻し手順を表している。進行中の処理として、ステップ１６０５で「第１」のスナップショットを作成する。この第１のスナップショットは、宛先上の複製スナップショットのエキスポートされたスナップショットである場合がある。一時的に、ソースからの差分更新によって、対象の宛先アクティブファイルシステム（複製スナップショット）が変更される（ステップ１６１０）。

【００９９】

パニック、クラッシュ、更新失敗等の非常事態に応答して、ユーザが開始したコマンドを終了させるため、ロールバックが開始される（ステップ１６１５）。この条件は、複製スナップショットの次の差分更新が適切に行なわれなかった場合の他、データの正確な像を反映しなかった場合などである。

【０１００】

ロールバックの開始に応じて、複製スナップショットに対するさらなる変更／更新が、停止される（ステップ１６２０）。これによって、アクティブファイルシステムを停止直後に次のステップ（下記ステップ１６２５）でアクティブファイルシステムから作成される第２のスナップショットに反映すべき状態とは異なる状態にする可能性がある、さらなる変更を回避することができる。アクティブファイルシステムに対する変更は、ファイルシステムを読み出し専用状態にしたりすべてのアクセスを拒否させる等、様々な方法を用いて停止させることができる。一実施形態では、「ＳＹＳＴＥＭ　ＡＮＤ　ＭＥＴＨＯＤ　ＦＯＲ　ＲＥＤＩＲＥＣＴＩＮＧ　ＡＣＣＥＳＳ　ＴＯ　Ａ　ＲＥＭＯＴＥ　ＭＩＲＲＯＲＥＤ　ＳＮＡＰＳＨＯＴ」と題した上記米国特許出願に記載されているように、アクティブファイルシステムのｉｎｏｄｅルックアップのために二次的な階層を導入することによって、アクティブファイルシステムに対するアクセスをエキスポートされたスナップショットにリダイレクトしている。

【０１０１】

停止させた後、次に、変更されたアクティブファイルシステムのうちの最も新しい状態にある「第２」のエキスポートされたスナップショットを作成する（ステップ１６２５）。

【０１０２】

次にステップ１６３０で、第２のスナップショットと第１のスナップショットの変更差分を計算する。これは、図８及び図９を参照した上記手順に従って行ない、第２のスナップショットをベースとして用い、第１のスナップショットを差分として用いる。次にステップ１６３５で、計算された変更差分をアクティブファイルシステム（現在の状態で凍結さ

れている）に適用する。この変更は、アクティブファイルシステムが最終的に第１のスナップショットに保持された状態に「巻き戻」されるように適用される（ステップ１６４０）。これが、巻き戻す必要がある緊急事態の前のアクティブファイルシステムの状態である。

【０１０３】

特定の状況では、ステップ１６２５によるアクティブファイルシステムのさらなる更新に対する停止や凍結を解除し、アクティブファイルシステムを変更やユーザ介入に対して再びアクセス可能にする場合もある（ステップ１６４５）。しかしながら、巻き戻し（下記）等の特定処理の場合、巻き戻されたＱｔｒｅｅは、複製処理による更なる変更についての制御下で維持される。

【０１０４】

この実施形態による巻き戻しの利点は、個別のログを維持したり、多量のシステムリソースを消費したりすることなく、複製されたデータ集合に対する一連の変更を元に戻すことが可能になるという点である。さらに、巻き戻しの方向、すなわち過去から現在であるか現在から過去であるかが、ほとんど問題にならない。さらに、一時ディレクトリを使用し、ファイルを消去しないことにより、この巻き戻しは、既存のＮＦＳクライアントに悪影響を与えることがない。各ＮＦＳクライアントは、ｉｎｏｄｅ番号とそのファイルの生成とを含むファイルハンドルによって、ファイルにアクセスする。システムがファイルを消去して再作成する場合、そのファイルは別のｉｎｏｄｅ／生成タプルを有することになる。そのため、ＮＦＳクライアントは、ファイルをリロード（古くなったファイルハンドルに関するメッセージを見る）することなくそのファイルにアクセスすることができない。しかしながら、一時ディレクトリによって、リンクの切れたファイルは、転送の終了時まで待たせることできる。そのため、上記の巻き戻しは、ＮＦＳクライアントが通知を出すことなく、一時ディレクトリに移動されただけのファイルを復活させることができる。

【０１０５】

ｉｎｏｄｅマップ反転

宛先複製スナップショットは、例えばソースＱｔｒｅｅスナップショットを再構築するためにソースでも必要になる場合があり（換言すれば、ソースと宛先の役割が逆になる）、一般的な巻き戻しを利用するためには、ソースと宛先の間にｉｎｏｄｅマップが適切に関連付けられている必要がある。この理由は、それらの個々のツリーにおいて、ソースｉｎｏｄｅが宛先ｉｎｏｄｅと一致しないからである。同じ理由で宛先ツリーを構築するためにｉｎｏｄｅマップが使用され、巻き戻しの間、ソースは、宛先から戻ってきたあらゆるｉｎｏｄｅの性質を判定するためにマッピングを用いる必要がある。しかしながら、宛先のｉｎｏｄｅマップは、ソースが使用するのに便利な形態で情報が有効に索引付けされていない。むしろ、ソースは、適当な値を得るために、マップに提示された順番でランダムに探しまわらなければならない場合がある。

【０１０６】

ソース中心のｉｎｏｄｅマップを生成する一方法は、マップエントリの「反転」である。図１７は、この反転を実施する手順１７００を詳細に示したものである。反転処理は、他の理由に関する災害復旧手順の一部として開始された巻き戻しの一部として（自動的に、又はユーザ指示で）開始される（ステップ１７０５）。次に、宛先及びソースは、ネゴシエーションをしてｉｎｏｄｅマップファイルを宛先からソースに転送する。このネゴシエーションは、適当な誤り訂正及び受領応答を含む既知の転送方法で行なうことができる（ステップ１７１０）。これにより、ｉｎｏｄｅが宛先からソースに転送されて格納される。

【０１０７】

次に、ソースは、各ｉｎｏｄｅについて１つのエントリを有する空のｉｎｏｄｅマップファイルを、ソースＱｔｒｅｅに作成する（ステップ１７１５）。次いで、新たな宛先は、カウンタをＮ＝１（この例の場合）で初期化する（ステップ１７２０）。Ｎは、新たな宛先Ｑｔｒｅｅ上のｉｎｏｄｅ数を表す変数である。

【０１０８】
ステップ１７２５で、新たな宛先は、格納しているｉｎｏｄｅマップファイル（すなわち、古い宛先／新たなソースからのマップ）における古い宛先に関連付けられたエントリの中から、Ｎ番目のｉｎｏｄｅを探しだす。次に、新たな宛先は、そのようなエントリが存在するか否かを判定する（判断ステップ１７３０）。エントリが存在しない場合、新たなソース（古い宛先）のＮ番目のｉｎｏｄｅが割り当てられていないことを表すゼロエントリを、新たなｉｎｏｄｅマップファイルに作成する。しかしながら、新たなソース／古い宛先のＮ番目のｉｎｏｄｅが存在した場合、判断ステップ１７３０はステップ１７４０に分岐し、新たなｉｎｏｄｅマップファイル（ステップ１７１５で作成されたもの）に新しいエントリを作成する。この新たなエントリは、新たなソース（古い宛先）のＮ番目のｉｎｏｄｅを適切な新しい宛先（古いソース）ｉｎｏｄｅにマッピングする。代替の実施形態として、この新たなｉｎｏｄｅマップはマッピングを開始する前にマップの全てのフィールドをゼロエントリにしておくこともでき、その場合、「ゼロエントリ」の作成にはｉｎｏｄｅマップに配置された既存のゼロエントリを残しておくことも含まれると広く考えられる。

【０１０９】
次いで手順１７００は、Ｎが古い宛先ファイルシステムのｉｎｏｄｅ数に等しいか否かをチェックする（判断ステップ１７４５）。等しければ、ｉｎｏｄｅマップファイルが完成し、手順は終了する（ステップ１７５０）。逆に、マッピングすべきｉｎｏｄｅがさらに存在する場合、カウンタを１だけインクリメントする（ステップ１７５５でＮ＝Ｎ＋１）。同様に、新たなｉｎｏｄｅマップにゼロエントリが作成された場合も、手順１７００は判断ステップ１７４５に分岐し、カウンタをインクリメント（ステップ１７５５）するか、終了するか（ステップ１７５０）を判断する。ステップ１７５５でカウンタがインクリメントされた場合、手順はステップ１７２５に戻り、そこでインクリメントされたＮ番目のｉｎｏｄｅを探す。

【０１１０】
例えば、図１８は、３つの例示的エントリ１８０２、１８０４及び１８０６を順番に含む、例示的な古い宛先ｉｎｏｄｅマップファイル１８００を示すものである。フィールド１４０４、１４０６（ソースｉｎｏｄｅ番号及び宛先ｉｎｏｄｅ番号）、１４０８、１４１０（ソース生成番号及び宛先生成番号）については、図１４を参照して上で説明した。エントリ１８０２は、（古い）ソースｉｎｏｄｅ７２が（古い）宛先ｉｎｏｄｅ６０５にマッピングされることを示している。同様に、エントリ１８０４はソースｉｎｏｄｅ８３を宛先ｉｎｏｄｅ３２８にマッピングし、エントリ１８０６はソースｉｎｏｄｅ１９０を宛先ｉｎｏｄｅ１５０にマッピングする。

【０１１１】
図１９は、反転手順１７００に従って、図１８の古いｉｎｏｄｅマップファイル１８００から生成された例示的な新しいｉｎｏｄｅマップファイル１９００を示すものである。この新たなマップファイルには、新たなソース（古い宛先）ｉｎｏｄｅ１９０２、新たな宛先（古いソース）ｉｎｏｄｅ１９０４、新たなソース（古い宛先）生成番号１９０６、及び、新たな宛先（古いソース）生成番号１９０８についてのフィールドが含まれる。反転の結果、新たなソースｉｎｏｄｅ１５０に関するエントリ１９１０は、適当な索引順番に存在し、新たな宛先ｉｎｏｄｅ１９０（および生成番号）とペアになる。次に（一連の連続した、間にある新たなソースｉｎｏｄｅ１５１～３７２に関するエントリ１９１４の後）、新たなソースｉｎｏｄｅ３２８に関するエントリ１９１２は、新たな宛先ｉｎｏｄｅ８３にマッピングされる。同様に、間にある新たなソースｉｎｏｄｅ３２９～６０４に関するエントリ１９１８の後、新たなソースｉｎｏｄｅ６０５に関するエントリ１９１６が新たな宛先ｉｎｏｄｅ７２にマッピングされる。間にあるソースｉｎｏｄｅは、他の新しく存在する宛先ｉｎｏｄｅへのマッピングを有する場合もあるし、あるいは、新たなソースｉｎｏｄｅ６０６に関するエントリ１９３０に示すように（記憶している古いソースｉｎｏｄｅマップ（１８００）上に新たな宛先ｉｎｏｄｅがなにも検出されなかった場合に

、手順１７００のステップ１７３５で設けられるように）、ゼロ値を有する場合もある。

【０１１２】

Ｇ．ｉｎｏｄｅマップ関連

同一ソースの２つの複製／ミラースナップショットは、互いにミラー関係を確立することができる。これら２つのスナップショットは、元のソースに関する２つの異なる時点での表現である。図２０は、元のソース２００１が２つの複製／ミラースナップショット、すなわち宛先スナップショットＡ（２００２）及び宛先スナップショットＢ（２００４）を生成する、一般的な環境２０００を示すものである。各宛先スナップショットＡ及びＢ（２００２及び２００４）は、元のソース２００１から転送されたデータストリームのｉｎｏｄｅをマッピングするのに用いられる、ｉｎｏｄｅマップＡ及びＢ（２０１２及び２０１４の各々）に関連している。

【０１１３】

図示の例は、宛先スナップショットＡ（２００２）が、変更を転送して宛先スナップショットＢ（２００４）にミラーを作成するように、準備しているところである。しかしながら、逆の場合、すなわち宛先スナップショットＢがミラーを宛先スナップショットＢに作成する場合も考えられる。従って、宛先スナップショットＡ（２００２）は、宛先スナップショットＡからの複製データに対して所望の宛先システムとして機能する宛先スナップショットＢ（２００４）に転送を行なう際、新たなソースになる。上記の反転実施形態のように、新たなソース２００２は、そのｉｎｏｄｅマップＡ２０１２を宛先システム２００４に転送する。次いで宛先システム２００４は、２つのシステムのｉｎｏｄｅ間の関係を判定する。この場合、新たなソースシステム及び新たな宛先システムはいずれも、古いソース２００１の索引を外して各々のツリーを参照するようにした、各々のｉｎｏｄｅマップＡ及びＢ（２０１２及び２０１４）を有する。２つの個別ｉｎｏｄｅマップが存在する場合、「連携」プロセス２０１６はそれらのｉｎｏｄｅマップをｉｎｏｄｅづづ同時に調べる。このプロセスは、元のソース２００１の各ｉｎｏｄｅについて、ｉｎｏｄｅマップＡ及びＢの各々から「宛先ｉｎｏｄｅ／生成番号」を取り出す。次いでこのプロセスは、その新たなソースを、新たに関連づけられたｉｎｏｄｅマップ２０１８に関する適当なマップ索引として扱う。関連マップは、新たなソース索引番号に関する新たなソース生成番号が格納され、さらに各索引エントリがｉｎｏｄｅマップＢ（２０１４）から取り出した新たな宛先ｉｎｏｄｅ／生成番号に関連付け／マッピングされる。この新たなマップは、上記の原理に従って新たな宛先２００４で使用され、様々な時点に関する新たなソースでの変更に基づいて、そのディレクトリにツリーが構築される。

【０１１４】

例えば、マップＡでは古いソースＯＳｉｎｏｄｅ番号５５（ＯＳ５５）が古い宛先スナップショットＡのｉｎｏｄｅ８７（Ａ８７）にマッピングされ、マップＢではＯＳ５５が古い宛先Ｂのｉｎｏｄｅ９９（Ｂ９９）にマッピングされている場合を仮定する。Ｂを新たな宛先にし、Ａを新たなソースにするためには、共通索引ＯＳ５５に基づいて各々のマップからＡ８７及びＢ９９を取り出すプロセスを用いて、関連マップを作成する。関連マップには、新たなソースエントリ／新たな宛先エントリ　８７／９９が格納される。このマップには、古いマップＡ及びＢからのそれらの値を有する関連生成番号も含まれる。この手順は２つの古い宛先システムに適用されているが、本明細書に記載される方法に従う様々な方法により、３以上のシステムに適用することも可能であると考えられる。

【０１１５】

上記が本発明の例示的実施形態の詳細な説明である。本発明の思想及び範囲から外れることなく様々な変更及び追加を行なうことができる。例えば、図示の相互接続されたソースサーバ及び／又は宛先サーバの数は、異なる数にしてもよい。実際、ソースサーバと宛先サーバは同一マシンにすることもできる。複数のソースがデータを宛先に転送することができ、またその逆可能であると広く考えられる。同様に、サーバの内部アーキテクチャやそれらのストレージアレイ、並びに、ネットワーク接続及びプロトコルは、すべて大きく異るものでよい。様々なソースサーバ及び宛先サーバ上で用いられるオペレーティングシ

ステムも異なっていてよい。さらに、本明細書に記載されるオーレーティングシステム及び手順はいずれも、ハードウェア、ソフトウェア、コンピュータ上で実行されるプログラム命令を有するコンピュータ読み取り可能な媒体、あるいは、ハードウェアとソフトウェアの組み合わせで実施することができる。

【図面の簡単な説明】

【図１】従来の実施形態によるネットワークを介したソースファイルサーバから宛先ファイルサーバへのボリュームスナップショットの遠隔ミラーリングを示す略ブロック図である。

【図２】従来の実施形態により図１の相違検出器がソースファイルサーバから宛先ファイルサーバへブロックの変更を送信すべきか否かを判定するために用いる判断テーブルである。

【図３】本発明の原理を実施するソースファイルサーバ及び宛先ファイルサーバを含む、例示的ネットワーク及びファイルサーバ環境を表す略ブロック図である。

【図４】図３のファイルサーバで使用される例示的ストレージオペレーティングシステムを示す略ブロック図である。

【図５】例示的ファイルシステムｉｎｏｄｅ構造を示す略ブロック図である。

【図６】スナップショットｉｎｏｄｅを含む、図５の例示的ファイルシステムｉｎｏｄｅ構造を示す略ブロック図である。

【図７】データブロックが書き替えられた後の図６の例示的ファイルシステムｉｎｏｄｅ構造を示す略ブロック図である。

【図８】スナップショットミラーリングプロセスのソースでの例示的処理を示す略ブロック図である。

【図８Ａ】図８のスナップショットミラーリングプロセスにおいてｉｎｏｄｅ採集器プロセスと共に用いられる判断テーブルである。

【図８Ｂ】図８Ａのｉｎｏｄｅ採集器プロセスを示す例示的なベーススナップショットブロック及び差分スナップショットブロックを示す詳細な概略図である。

【図９】図８のスナップショットミラーリングプロセスとともに用いられるｉｎｏｄｅワーカーの例示的処理を示す略ブロック図である。

【図１０】ソースファイルサーバスナップショットミラーリングプロセス及び宛先スナップショットミラーリングプロセス、並びに、それらの間の通信リンクを示す略ブロック図である。

【図１１】例示的実施形態によるソースと宛先との間のデータストリーム伝送フォーマットに用いられる単独ヘッダ構造を示す略ブロック図である。

【図１２】例示的実施形態によるソースと宛先との間のデータストリーム伝送フォーマットを示す略ブロック図である。

【図１３】宛先におけるスナップショットミラーリングプロセスのステージを示す略ブロック図である。

【図１４】例示的実施形態によりソースｉｎｏｄｅを宛先スナップショットミラーにマッピングするための一般的なｉｎｏｄｅマップを示す略ブロック図である。

【図１５】宛先スナップショットミラーのソースデータファイルに関連してマッピングされたオフセットへのデータファイルの格納を示す極めて概略的な図である。

【図１６】例示的実施形態によるスナップショット巻き戻し手順を示すフロー図である。

【図１７】例示的実施形態によりソースファイルシステムを宛先ミラースナップショットの状態に巻き戻す、すなわち同期させるための、ｉｎｏｄｅマップ反転手順を示すフロー図である。

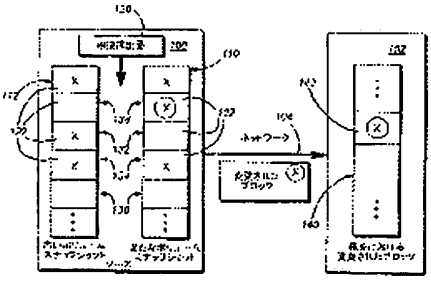
【図１８】図１７の反転手順で使用される、宛先にある例示的ｉｎｏｄｅマップを示す略ブロック図である。

【図１９】図１８の反転手順により古いソース（新たな宛先）上に構築された例示的ｉｎｏｄｅマップを示す略ブロック図である。

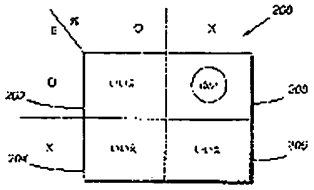
【図２０】例示的実施形態による一般的ないのでマップ関連付けプロセスを示す略ブロッ

ク図である。

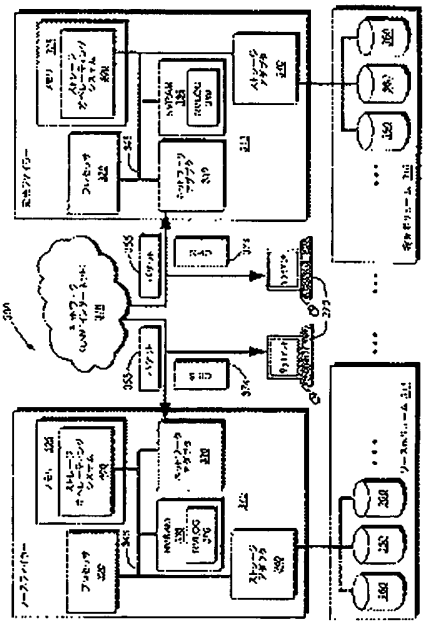
【図１】



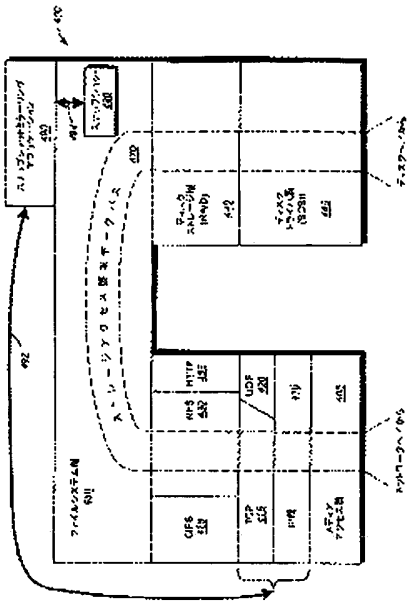
【図２】



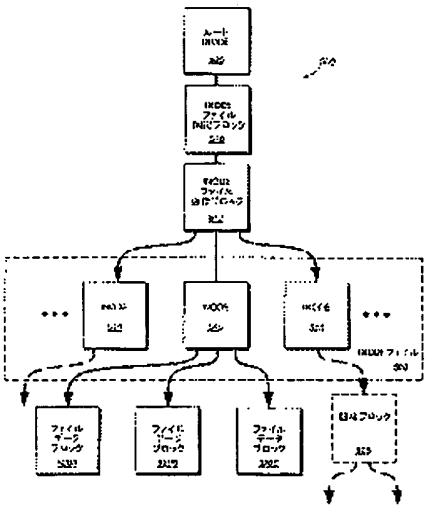
【図３】



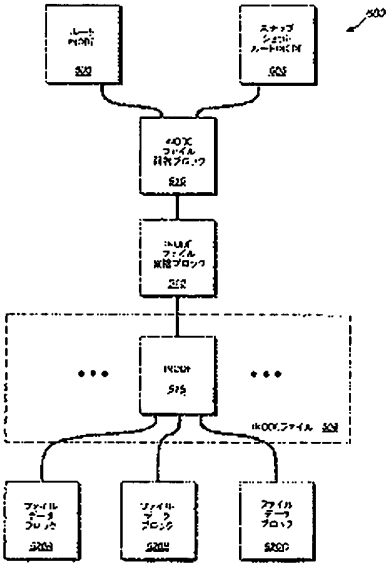
【図４】



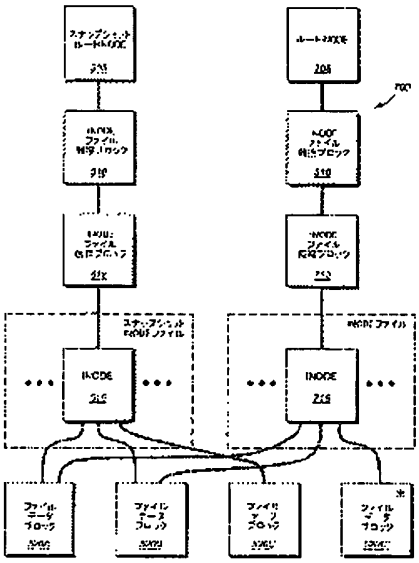
【図５】



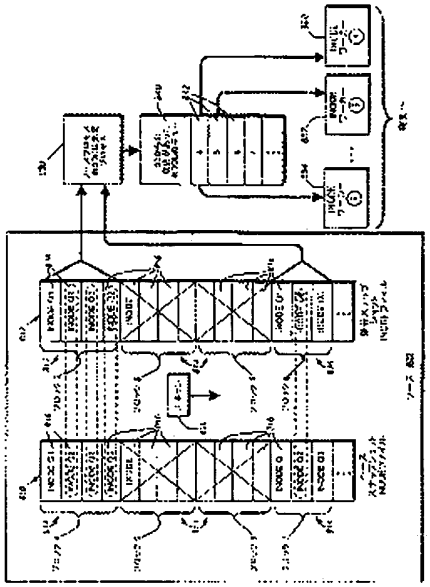
【図６】



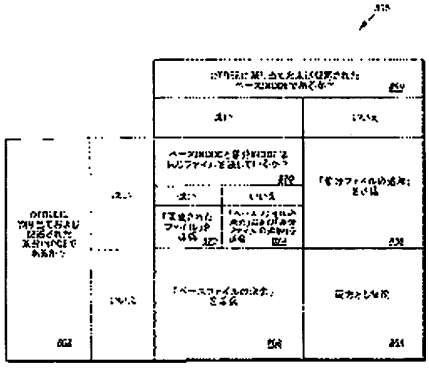
【図７】



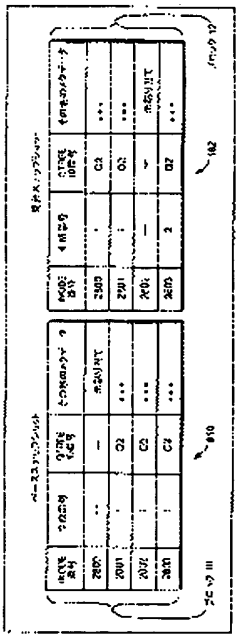
【図８】



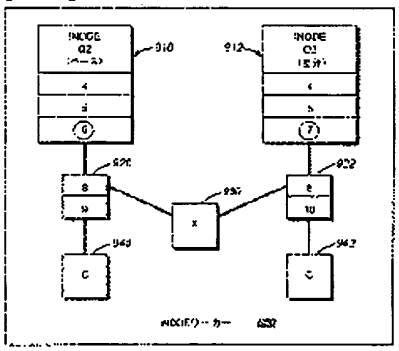
【図８Ａ】



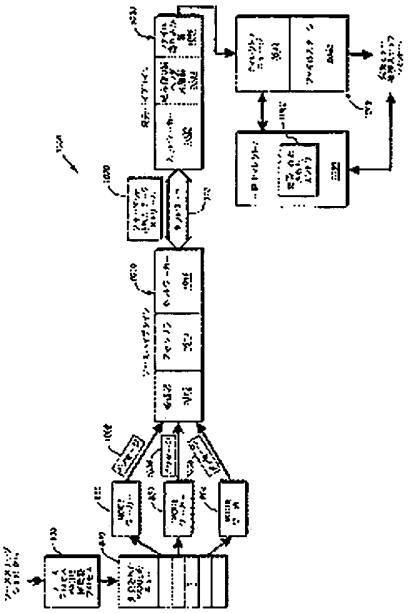
【図８Ｂ】



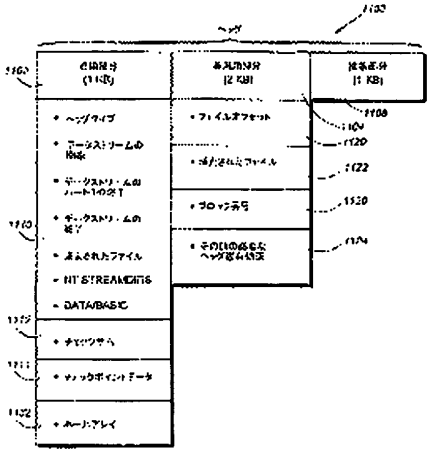
【図９】



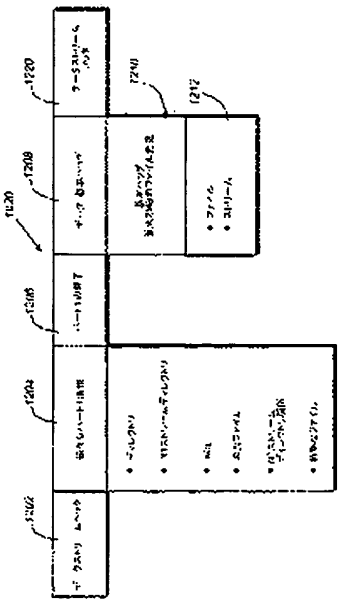
【図１０】



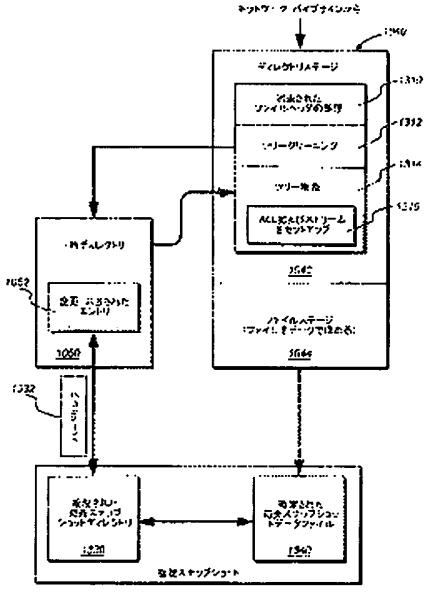
【図１１】



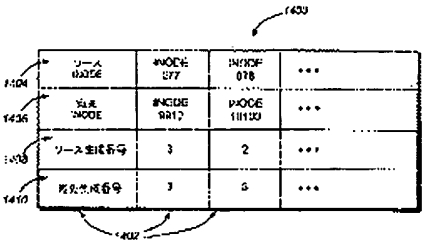
【図１２】



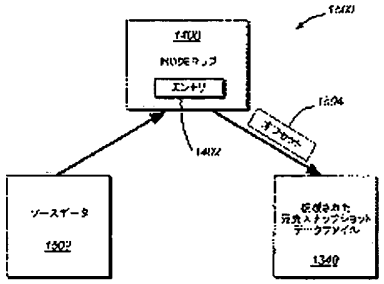
【図１３】



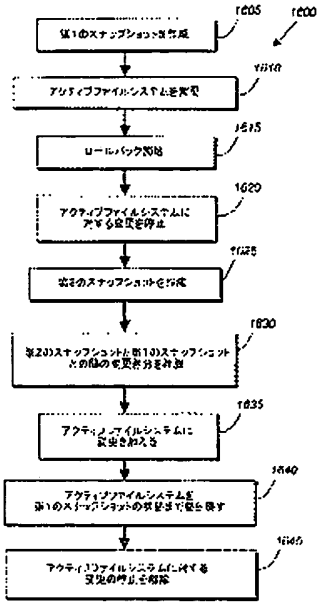
【図１４】



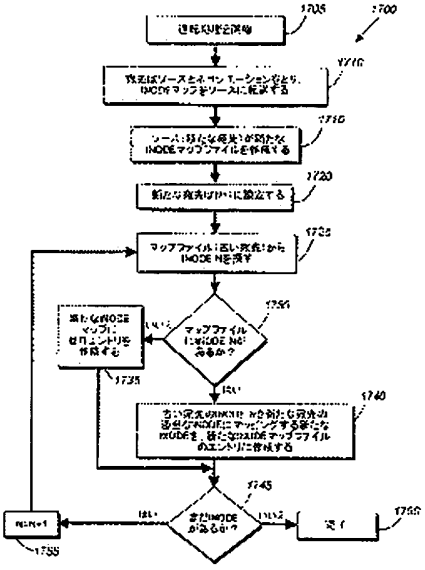
【図１５】



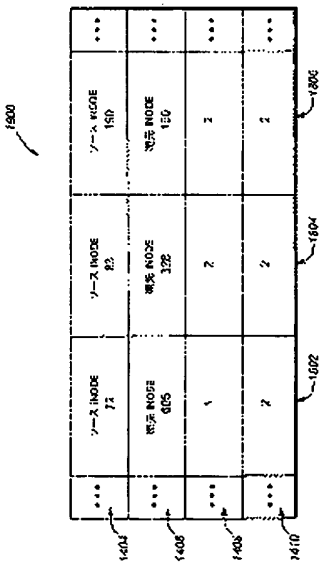
【図１６】



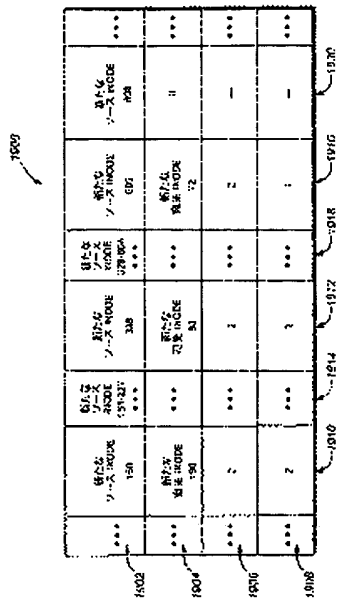
【図１７】



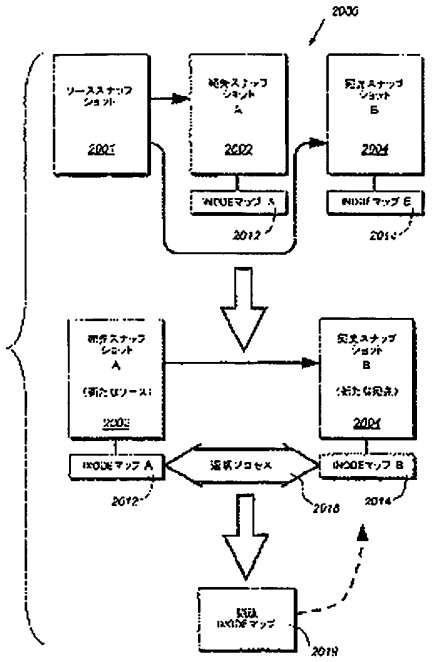
【図１８】



【図１９】



【図２０】



フロントページの続き

(72)発明者 マイケル・エル・フェダーウィッシュ
アメリカ合衆国カリフォルニア州９５１２５、サンノゼ、スレイシャー・レーン・２７４２
(72)発明者 シェーン・エス・オワラ
アメリカ合衆国カリフォルニア州９４０４０、マウンテンビュー，ゴルフ・コート・１０１０
(72)発明者 ステファン・エル・マンレイ
イギリス国・エスダブリュー５９ジェイゼット・ロンドン、アードレイ・クレセント・５４、フラット・４
(72)発明者 スティーブン・アール・クレイマン
アメリカ合衆国カリフォルニア州９４０８９、サニーベイル，イースト・ジャバ・ドライブ・４９５

Ｆターム(参考) 5B065 BA01 EA31 EK05
5B082 DE05 GA04 HA03

【要約の続き】
れた複製スナップ・ショットテ・ィレクトリに再リンクされる。
【選択図】 図１０

【外国語明細書】

SYSTEM AND METHOD FOR DETERMINING CHANGES IN TWO SNAPSHOTS AND FOR TRANSMITTING CHANGES TO A DESTINATION SNAPSHOT

RELATED APPLICATIONS

This application is related to the following United States Patent Applications:

Serial No. [Attorney Docket No. 112056-0063], entitled SYSTEM AND METHOD FOR ASYNCHRONOUS MIRRORING OF SNAPSHOTS AT A DESTINATION USING A PURGATORY DIRECTORY AND INODE MAPPING, by Stephen L. Manley, *et al.*, the teachings of which are expressly incorporated herein by reference;

Serial No. [Attorney Docket No. 112056-0052], entitled SYSTEM AND METHOD FOR STORAGE OF SNAPHOT METADATA IN A REMOTE FILE, by Stephen L. Manley, *et al.*, the teachings of which are expressly incorporated herein by reference;

Serial No. [Attorney Docket No. 112056-0053], entitled SYSTEM AND METHOD FOR REDIRECTING ACCESS TO A REMOTE MIRRORED SNAPSHOT, by Raymond C. Chen, *et al.*, the teachings of which are expressly incorporated herein by reference;

Serial No. [Attorney Docket No. 112056-0062], entitled FORMAT FOR TRANSMISSION OF FILE SYSTEM INFORMATION BETWEEN A SOURCE AND A DESTINATION, by Stephen L. Manley, *et al.*, the teachings of which are expressly incorporated herein by reference; and

Serial No. [Attorney Docket No. 112056-0055], entitled SYSTEM AND METHOD FOR CHECKPOINTING AND RESTARTING AN ASYNCHRONOUS TRANSFER OF DATA BETWEEN A SOURCE AND DESTINATION SNAPSHOT, by Michael L. Federwisch, *et al.*, the teachings of which are expressly incorporated herein by reference.

FIELD OF THE INVENTION

This invention relates to storage of data using file servers and more particularly to mirroring or replication of stored data in remote storage locations over a network.

BACKGROUND OF THE INVENTION

A file server is a computer that provides file service relating to the organization of information on storage devices, such as disks. The file server or *filer* includes a storage operating system that implements a file system to logically organize the information as a hierarchical structure of directories and files on the disks. Each "on-disk" file may be implemented as a set of data structures, e.g., disk blocks, configured to store information. A directory, on the other hand, may be implemented as a specially formatted file in which information about other files and directories are stored.

A filer may be further configured to operate according to a client/server model of information delivery to thereby allow many clients to access files stored on a server, e.g., the filer. In this model, the client may comprise an application, such as a database application, executing on a computer that "connects" to the filer over a direct connection or computer network, such as a point-to-point link, shared local area network (LAN), wide area network (WAN), or virtual private network (VPN) implemented over a public network such as the Internet. Each client may request the services of the file system on the filer by issuing file system protocol messages (in the form of packets) to the filer over the network.

A common type of file system is a "write in-place" file system, an example of which is the conventional Berkeley fast file system. By "file system" it is meant generally a structuring of data and metadata on a storage device, such as disks, which permits reading/writing of data on those disks. In a write in-place file system, the locations of the data structures, such as inodes and data blocks, on disk are typically fixed. An inode is a data structure used to store information, such as metadata, about a file, whereas the data blocks are structures used to store the actual data for the file. The information contained in an inode may include, e.g., ownership of the file, access permission for the file, size of the file, file type and references to locations on disk of the data blocks for the file. The

references to the locations of the file data are provided by pointers in the inode, which may further reference indirect blocks that, in turn, reference the data blocks, depending upon the quantity of data in the file. Changes to the inodes and data blocks are made "in-place" in accordance with the write in-place file system. If an update to a file extends the quantity of data for the file, an additional data block is allocated and the appropriate inode is updated to reference that data block.

Another type of file system is a write-anywhere file system that does not overwrite data on disks. If a data block on disk is retrieved (read) from disk into memory and "dirtied" with new data, the data block is stored (written) to a new location on disk to thereby optimize write performance. A write-anywhere file system may initially assume an optimal layout such that the data is substantially contiguously arranged on disks. The optimal disk layout results in efficient access operations, particularly for sequential read operations, directed to the disks. A particular example of a write-anywhere file system that is configured to operate on a filer is the Write Anywhere File Layout (WAFL™) file system available from Network Appliance, Inc. of Sunnyvale, California. The WAFL file system is implemented within a microkernel as part of the overall protocol stack of the filer and associated disk storage. This microkernel is supplied as part of Network Appliance's Data ONTAP™ software, residing on the filer, that processes file-service requests from network-attached clients.

As used herein, the term "storage operating system" generally refers to the computer-executable code operable on a computer that manages data access and may, in the case of a filer, implement file system semantics, such as the Data ONTAP™ storage operating system, implemented as a microkernel, and available from Network Appliance, Inc. of Sunnyvale, California, which implements a Write Anywhere File Layout (WAFL™) file system. The storage operating system can also be implemented as an application program operating over a general-purpose operating system, such as UNIX® or Windows NT®, or as a general-purpose operating system with configurable functionality, which is configured for storage applications as described herein.

Disk storage is typically implemented as one or more storage "volumes" that comprise physical storage disks, defining an overall logical arrangement of storage space.

Currently available filer implementations can serve a large number of discrete volumes (150 or more, for example). Each volume is associated with its own file system and, for purposes hereof, volume and file system shall generally be used synonymously. The disks within a volume are typically organized as one or more groups of Redundant Array of Independent (or *Inexpensive*) Disks (RAID). RAID implementations enhance the reliability/integrity of data storage through the redundant writing of data "stripes" across a given number of physical disks in the RAID group, and the appropriate caching of parity information with respect to the striped data. In the example of a WAFL file system, a RAID 4 implementation is advantageously employed. This implementation specifically entails the striping of data across a group of disks, and separate parity caching within a selected disk of the RAID group. As described herein, a *volume* typically comprises at least one data disk and one associated parity disk (or possibly data/parity partitions in a single disk) arranged according to a RAID 4, or equivalent high-reliability, implementation.

In order to improve reliability and facilitate disaster recovery in the event of a failure of a filer, its associated disks or some portion of the storage infrastructure, it is common to "mirror" or replicate some or all of the underlying data and/or the file system that organizes the data. In one example, a mirror is established and stored at a remote site, making it more likely that recovery is possible in the event of a true disaster that may physically damage the main storage location or it's infrastructure (e.g. a flood, power outage, act of war, etc.). The mirror is updated at regular intervals, typically set by an administrator, in an effort to catch the most recent changes to the file system. One common form of update involves the use of a "snapshot" process in which the active file system at the storage site, consisting of inodes and blocks, is captured and the "snapshot" is transmitted as a whole, over a network (such as the well-known Internet) to the remote storage site. Generally, a snapshot is an image (typically read-only) of a file system at a point in time, which is stored on the same primary storage device as is the active file system and is accessible by users of the active file system. By "active file system" it is meant the file system to which current input/output operations are being directed. The primary storage device, e.g., a set of disks, stores the active file system, while a secondary storage, e.g. a tape drive, may be utilized to store backups of the active file system.

Once snapshotted, the active file system is reestablished, leaving the snapshotted version in place for possible disaster recovery. Each time a snapshot occurs, the old active file system becomes the new snapshot, and the new active file system carries on, recording any new changes. A set number of snapshots may be retained depending upon various time-based and other criteria. The snapshotting process is described in further detail in United States Patent Application Serial No. 09/932,578, entitled INSTANT SNAPSHOT by Blake Lewis *et al.*, which is hereby incorporated by reference as though fully set forth herein. In addition, the native Snapshot™ capabilities of the WAFL file system are further described in *TR3002 File System Design for an NFS File Server Appliance* by David Hitz *et al.*, published by Network Appliance, Inc., and in commonly owned U.S. Patent No. 5,819,292 entitled METHOD FOR MAINTAINING CONSISTENT STATES OF A FILE SYSTEM AND FOR CREATING USER-ACCESSIBLE READ-ONLY COPIES OF A FILE SYSTEM by David Hitz *et al.*, which are hereby incorporated by reference.

The complete recopying of the entire file system to a remote (destination) site over a network may be quite inconvenient where the size of the file system is measured in tens or hundreds of gigabytes (even terabytes). This full-backup approach to remote data replication may severely tax the bandwidth of the network and also the processing capabilities of both the destination and source filer. One solution has been to limit the snapshot to only portions of a file system volume that have experienced changes. Hence, Fig. 1 shows a prior art volume-based mirroring where a source file system 100 is connected to a destination storage site 102 (consisting of a server and attached storage—not shown) via a network link 104. The destination 102 receives periodic snapshot updates at some regular interval set by an administrator. These intervals are chosen based upon a variety of criteria including available bandwidth, importance of the data, frequency of changes and overall volume size.

In brief summary, the source creates a pair of time-separated snapshots of the volume. These can be created as part of the commit process in which data is committed to non-volatile memory in the filer or by another mechanism. The "new" snapshot 110 is a recent snapshot of the volume's active file system. The "old" snapshot 112 is an older snapshot of the volume, which should match the image of the file system replicated on

the destination mirror. Note, that the file server is free to continue work on new file service requests once the new snapshot 112 is made. The new snapshot acts as a checkpoint of activity up to that time rather than an absolute representation of the then-current volume state. A differencer 120 scans the blocks 122 in the old and new snapshots. In particular, the differencer works in a block-by-block fashion, examining the list of blocks in each snapshot to compare which blocks have been allocated. In the case of a write-anywhere system, the block is not reused as long as a snapshot references it, thus a change in data is written to a new block. Where a change is identified (denoted by a presence or absence of an 'X' designating data), a decision process 200, shown in Fig. 2, in the differencer 120 decides whether to transmit the data to the destination 102. The process 200 compares the old and new blocks as follows: (a) Where data is in neither an old nor new block (case 202) as in old/new block pair 130, no data is available to transfer. (b) Where data is in the old block, but not the new (case 204) as in old/new block pair 132, such data has already been transferred, (and any new destination snapshot pointers will ignore it), so the new block state is not transmitted. (c) Where data is present in the both the old block and the new block (case 206) as in the old/new block pair 134, no change has occurred and the block data has already been transferred in a previous snapshot. (d) Finally, where the data is not in the old block, but is in the new block (case 208) as in old/new block pair 136, then a changed data block is transferred over the network to become part of the changed volume snapshot set 140 at the destination as a changed block 142. In the exemplary write-anywhere arrangement, the changed blocks are written to new, unused locations in the storage array. Once all changed blocks are written, a base file system information block, that is the root pointer of the new snapshot, is then committed to the destination. The transmitted file system information block is committed, and updates the overall destination file system by pointing to the changed block structure in the destination, and replacing the previous file system information block. The changes are at this point committed as the latest incremental update of the destination volume snapshot. This file system accurately represents the "new" snapshot on the source. In time a new "new" snapshot is created from further incremental changes.

Approaches to volume-based remote mirroring of snapshots are described in detail in commonly owned U.S. Patent Application Serial No. 09/127,497, entitled FILE SYSTEM IMAGE TRANSFER by Steven Kleiman, *et al.* and U.S. Patent Application Serial No. 09/426,409, entitled FILE SYSTEM IMAGE TRANSFER BETWEEN DISSIMILAR FILE SYSTEMS by Steven Kleiman, *et al.*, both of which patents are expressly incorporated herein by reference.

This volume-based approach to incremental mirroring from a source to a remote storage destination is effective, but may still be inefficient and time-consuming as it forces an entire volume to be scanned for changes and those changes to be transmitted on a block-by-block basis. In other words, the scan focuses on blocks without regard to any underlying information about the files, inodes and data structures, which the blocks comprise. The destination is organized as a set of volumes so a direct volume-by-volume mapping is established between source and destination. Again, where a volume may contain a terabyte or more of information, the block-by-block approach to scanning and comparing changes may still involve significant processor overhead and associated processing time. Often, there may have been only minor changes in a sub-block beneath the root inode block being scanned. Since a list of all blocks in the volume is being examined, however, the fact that many groupings of blocks (files, inode structures, etc.) are unchanged is not considered. In addition, the increasingly large size and scope of a full volume make it highly desirable to sub-divide the data being mirrored into sub-groups, because some groups are more likely to undergo frequent changes, it may be desirable to update their replicas more often than other, less-frequently changed groups. In addition, it may be desirable to mingle original and replicated (snapshotted) sub-groups in a single volume and migrate certain key data to remote locations without migrating an entire volume. Accordingly, a more sophisticated approach to scanning and identifying changed blocks may be desirable, as well as a sub-organization for the volume that allows for the mirroring of less-than-an-entire volume.

One such sub-organization of a volume is the well-known qtree. Qtrees, as implemented on an exemplary storage system such as described herein, are subtrees in a volume's file system. One key feature of qtrees is that, given a particular qtree, any file or directory in the system can be quickly tested for membership in that qtree, so they serve as a good way to organize the file system into discrete data sets. The use of qtrees as a source and destination for snapshotted data is desirable.

When a qtree is snapshot is replicated at the destination, it is typically made available for disaster recovery and other uses, such as data distribution. However, the snapshot residing on the destination's active file system may be in the midst of receiving or processing an update from the source snapshot when access by a user or process is desired. A way to allow the snapshot to complete its update without interference is highly desirable. Likewise, when a snapshot must return to an earlier state, a way to efficiently facilitate such a return or "rollback" is desired. A variety of other techniques for manipulating different point in time snapshots may increase the versatility and utility of a snapshot replication mechanism.

In addition, the speed at which a destination snapshot may be updated is partially depends upon the speed whit which change data can be committed from the source to the destination's active file system. Techniques for improving the efficiency of file deletion and modification are also highly desirable.

SUMMARY OF THE INVENTION

This invention overcomes the disadvantages of the prior art by providing a system and method for remote asynchronous replication or mirroring of changes in a source file system snapshot in a destination replica file system using a scan (via a scanner) of the blocks that make up two versions of a snapshot of the source file system, which identifies changed blocks in the respective snapshot files based upon differences in volume block numbers identified in a scan of the logical file block index of each snapshot. Trees of blocks associated with the files are traversed, bypassing unchanged pointers between versions and walking down to identify the changes in the hierarchy of the tree. These changes are transmitted to the destination mirror or replicated snapshot. This technique allows regular files, directories, inodes and any other hierarchical structure to be efficiently scanned to determine differences between versions thereof.

According to an illustrative embodiment, the source scans, with the scanner, along the index of logical file blocks for each snapshot looking for changed volume block numbers between the two source snapshots. Since disk blocks are always rewritten to new locations on the disk, a difference indicates changes in the underlying inodes of the respective blocks. Using the scanner, unchanged blocks are efficiently overlooked, as their inodes are unchanged. Using an inode picker process, that receives changed blocks from the scanner the source picks out inodes from changed blocks specifically associated with the selected qtree (or other sub-organization of the volume). The picker process looks for versions of inodes that have changed between the two snapshots and picks out the changed version. If inodes are the same, but files have changed (based upon different generation numbers in the inodes) the two versions of the respective inodes are both picked out. The changed versions of the inodes (between the two snapshots) are queued and transferred to a set of inode handlers/workers or handlers that resolve the changes in underlying blocks by continuing to scan (with the scanner, again) file offsets down "trees" of the inodes until differences in underlying blocks are identified via their block pointers, as changed inodes in one version will point to different data blocks than those in

the other version. Only the *changes* in the trees are transmitted over the network for update of the destination file system in an asynchronous (lazy write) manner. The destination file system is exported read-only the user. This ensures that only the replicator can alter the state of the replica file system.

In an illustrative embodiment, a file system-independent format is used to transmit a data stream of change data over the network. This format consists of a set of standalone headers with unique identifiers. Some headers refer to follow-on data and others carry relevant data within their stream. For example, the information relating to any source snapshot deleted files are carried within "deleted files" headers. All directory activity is transmitted first, followed by file data. File data is sent in chunks of varying size, separated by regular headers until an ending header (footer) is provided. At the destination, the format is unpacked and inodes contained therein are transmitted over the network are mapped to a new directory structure. Received file data blocks are written according to their offset in the corresponding destination file. An inode map stores entries which map the source's inodes (files) to the destination's inodes (files). The inode map also contains generation numbers. The tuple of (inode number, generation number) allows the system to create a file handle for fast access to a file. It also allows the system to track changes in which a file is deleted and its inode number is reassigned to a newly created file. To facilitate construction of a new directory tree on the destination, an initial directory stage of the destination mirror process receives source directory information via the format and moves any deleted or moved files to a temporary or "purgatory" directory. The purgatory files which have been moved are hard linked from the purgatory directory to the directories where they have been moved to. Newly created source files are entered into map and built into the directory tree. After the directory tree is built, the transfer of file data begins. Changes to file data from the source are written to the corresponding replica files (as identified by the inode map). When the data stream transfer is complete, the purgatory directory is removed and any unlinked files (including various deleted files) are permanently deleted. In one embodiment, a plurality of discrete source qtrees or other sub-organizations derived from different source volumes can be replicated/mirrored on a single destination volume.

This invention overcomes the disadvantages of the prior art, in a system and method for updating a replicated destination file system snapshot with changes in a source file system snapshot, by facilitating construction of a new directory tree on the destination from source update information using a temporary or "purgatory" directory that allows any modified and deleted files on the destination active file system to be associated with (e.g. moved to) the purgatory directory if and until they are reused. In addition, an inode map is established on the destination that maps source inode numbers to destination inode numbers so as to facilitate building of the destination tree using inode/generation number *tuples*. The inode map allows resynchronization of the source file system to the destination. The inode map also allows association of two or more destination snapshots to each other based upon their respective maps with the source.

In an illustrative embodiment, a file system-independent format is used to transmit a data stream of changed file data blocks with respect to a source's base and incremental snapshots. Received file data blocks are written according to their offset in the corresponding destination file. An inode map stores entries which map the source's inodes (files) to the destination's inodes (files). The inode map also contains generation numbers. The tuple of (inode number, generation number) allows the system to create a file handle for fast access to a file. It also allows the system to track changes in which a file is deleted and its inode number is reassigned to a newly created file. To facilitate construction of a new directory tree on the destination, an initial directory stage of the destination mirror process receives source directory information via the format and moves any deleted or moved files to the temporary or "purgatory" directory. The purgatory files which have been moved are hard linked from the purgatory directory to the directories where they have been moved to. Newly created source files are entered into map and built into the directory tree. After the directory tree is built, the transfer of file data begins. Changes to file data from the source are written to the corresponding replica files (as identified by the inode map). When the data stream transfer is complete, the purgatory directory is removed and any unlinked files (including various deleted files) are permanently deleted. In one embodiment, a plurality of discrete source qtrees or other sub-organizations derived from different source volumes can be replicated/mirrored on a single destination volume.

In another illustrative embodiment, the replicated file system is, itself snapshotted, thereby creating a first exported snapshot. The first exported snapshot corresponds to a first state. If a disaster or communication breakdown occurs after further modifications or updates of the replicated snapshot, then further modification to the replica file system is halted/frozen and a subsequent second exported snapshot is created from the frozen replica file system representing the second state. The replicated file system can be "rolled back" from the second state to the first state by determining the differences in data between the second state and the first state and then applying those changes to recreate the first state.

In yet another illustrative embodiment, the inode map used to map inodes transferred from the source snapshot to inodes in the destination replica/mirror file system is used to resynchronize the source state with the destination state. The destination becomes a new "source" and negotiates the transfer of the inode map to the old "source" now the new "destination." The received old inode map is stored on the source and accessed by a flip procedure that generates a new destination map with N inodes equal to the number of inodes on the new source. The new destination then creates entries from the stored source map for each new destination associated with the new source entry if available. Associated generation numbers are also mapped, thereby providing the needed file access tuple. Any entries on the new source index that lack a new destination are marked as zero entries. The completed flipped inode map allows the new source to update the new destination with its changed data.

In a related embodiment, two replica/mirror snapshots of the same source can establish a mirror relationship with one another. As in the flip embodiment above, the new "source" (old replica) transfers its inode map to the destination system. The destination system then determines the relationship between the two system's inodes. An "associative" process walks the inode maps at the same time (e.g. concurrently, inode number-by-inode number). For each inode from the original source, the process extracts the "destination inode/generation" from each of the inode maps. It then treats the new source as the appropriate map index for the new inode map. It stores the new source generation number, as well as the destination inode/generation.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings in which like reference numerals indicate identical or functionally similar elements:

Fig. 1, already described, is a schematic block diagram of an exemplary remote mirroring of a volume snapshot from a source file server to a destination file server over a network according to a prior implementation;

Fig. 2, already described, is a decision table used by a block differencer of Fig. 1 for determining whether a change in a block is to be transmitted from the source file server to the destination file server according to a prior implementation;

Fig. 3 is a schematic block diagram defining an exemplary network and file server environment including a source file server and a destination file server within which the principles of this invention are implemented;

Fig 4 is a schematic block diagram of an exemplary storage operating system for use with the file servers of Fig. 3;

Fig. 5 is schematic block diagram of an exemplary file system inode structure;

Fig. 6 is a schematic block diagram of the exemplary file system inode structure of Fig. 5 including a snapshot inode;

Fig. 7 is a schematic block diagram of the exemplary file system inode structure of Fig. 6 after data block has been rewritten;

Fig. 8 is a schematic block diagram of an exemplary operation of the snapshot mirroring process at the source;

Fig. 8A is a decision table used in connection with an inode picker process in the snapshot mirroring process of Fig. 8;

Fig. 8B is a more detailed schematic diagram of an exemplary base snapshot and incremental snapshot block illustrating the inode picker process of Fig. 8A;

Fig. 9 is a schematic block diagram of an exemplary operation of an inode worker used in connection with the snapshot mirroring process of Fig. 8;

Fig. 10 is a schematic block diagram of the source file server snapshot mirroring process, the destination snapshot mirroring process, and the communication link between them;

Fig. 11 is a schematic block diagram of a standalone header structure for use in the data stream transmission format between the source and the destination according to an illustrative embodiment;

Fig. 12 is a schematic block diagram of the data stream transmission format between the source and the destination according to an illustrative embodiment;

Fig. 13 is a schematic block diagram of the stages of the snapshot mirroring process on the destination;

Fig. 14 is a schematic block diagram of a generalized inode map for mapping source inodes to the destination snapshot mirror according to an illustrative embodiment;

Fig. 15 is a highly schematic diagram of the population of data files in the destination snapshot mirror at mapped offsets with respect to source data files;

Fig. 16 is a flow diagram of a snapshot rollback procedure according to an illustrative embodiment; and

Fig. 17 is a flow diagram of a inode map flipping procedure for rolling back or resynchronizing the source file system to a state of the destination mirror snapshot according to an illustrative embodiment;

Fig. 18 is a schematic block diagram of an exemplary inode map residing on the destination for use in the flipping procedure of Fig. 17;

Fig. 19 is a schematic block diagram of an exemplary inode map constructed on the old source (new destination) according to the flipping procedure of Fig. 18;

Fig. 20 is a schematic block diagram of a generalized inode map association process according to an illustrative embodiment.

DETAILED DESCRIPTION OF AN ILLUSTRATIVE EMBODIMENT

A. Network and File Server Environment

By way of further background, Fig. 3 is a schematic block diagram of a storage system environment 300 that includes a pair of interconnected file servers including a source file server 310 and a destination file server 312 that may each be advantageously used with the present invention. For the purposes of this description, the source file server is a networked computer that manages storage one or more source volumes 314,

each having an array of storage disks 360 (described further below). Likewise, the destination filer 312 manages one or more destination volumes 316, also comprising arrays of disks 360. The source and destination file servers or "filers" are linked via a network 318 that can comprise a local or wide area network, such as the well-known Internet. An appropriate network adapter 330 residing in each filer 310, 312 facilitates communication over the network 318. Also for the purposes of this description, like components in each of the source and destination filer, 310 and 312 respectively, are described with like reference numerals. As used herein, the term "source" can be broadly defined as a location from which the subject data of this invention travels and the term "destination" can be defined as the location to which the data travels. While a source filer and a destination filer, connected by a network, is a particular example of a source and destination used herein, a source and destination could be computers/filers linked via a direct link, or via loopback (a "networking" arrangement internal to a single computer for transmitting a data stream between local source and local destination), in which case the source and the destination are the same filer. As will be described further below, the source and destination are broadly considered to be a source sub-organization of a volume and a destination sub-organization of a volume. Indeed, in at least one special case the source and destination sub-organizations can be the same at different points in time.

In the particular example of a pair of networked source and destination filers, each filer 310 and 312 can be any type of special-purpose computer (e.g., server) or general-purpose computer, including a standalone computer. The source and destination filers 310, 312 each comprise a processor 320, a memory 325, a network adapter 350 and a storage adapter 340 interconnected by a system bus 345. Each filer 310, 312 also includes a storage operating system 400 (Fig. 4) that implements a file system to logically organize the information as a hierarchical structure of directories and files on the disks.

It will be understood to those skilled in the art that the inventive technique described herein may apply to any type of special-purpose computer (e.g., file serving appliance) or general-purpose computer, including a standalone computer, embodied as a storage system. To that end, the filers 310 and 312 can each be broadly, and alternatively, referred to as storage systems. Moreover, the teachings of this invention can be adapted to a variety of storage system architectures including, but not limited to, a net-

work-attached storage environment, a storage area network and disk assembly directly-attached to a client/host computer. The term "storage system" should, therefore, be taken broadly to include such arrangements.

In the illustrative embodiment, the memory 325 comprises storage locations that are addressable by the processor and adapters for storing software program code. The memory comprises a form of random access memory (RAM) that is generally cleared by a power cycle or other reboot operation (i.e., it is "volatile" memory). The processor and adapters may, in turn, comprise processing elements and/or logic circuitry configured to execute the software code and manipulate the data structures. The operating system 400, portions of which are typically resident in memory and executed by the processing elements, functionally organizes the filer by, *inter alia*, invoking storage operations in support of a file service implemented by the filer. It will be apparent to those skilled in the art that other processing and memory means, including various computer readable media, may be used for storing and executing program instructions pertaining to the inventive technique described herein.

The network adapter 330 comprises the mechanical, electrical and signaling circuitry needed to connect each filer 310, 312 to the network 318, which may comprise a point-to-point connection or a shared medium, such as a local area network. Moreover the source filer 310 may interact with the destination filer 312 in accordance with a client/server model of information delivery. That is, the client may request the services of the filer, and the filer may return the results of the services requested by the client, by exchanging packets 355 encapsulating, e.g., the TCP/IP protocol or another network protocol format over the network 318.

The storage adapter 340 cooperates with the operating system 400 (Fig. 4) executing on the filer to access information requested by the client. The information may be stored on the disks 360 that are attached, via the storage adapter 340 to each filer 310, 312 or other node of a storage system as defined herein. The storage adapter 340 includes input/output (I/O) interface circuitry that couples to the disks over an I/O interconnect arrangement, such as a conventional high-performance, Fibre Channel serial link topology. The information is retrieved by the storage adapter and processed by the proc-

essor 320 as part of the snapshot procedure, to be described below, prior to being forwarded over the system bus 345 to the network adapter 330, where the information is formatted into a packet and transmitted to the destination server as also described in detail below.

Each filer may also be interconnected with one or more clients 370 via the network adapter 330. The clients transmit requests for file service to the source and destination filers 310, 312, respectively, and receive responses to the requests over a LAN or other network (318). Data is transferred between the client and the respective filer 310, 312 using data packets 374 defined as an encapsulation of the Common Internet File System (CIFS) protocol or another appropriate protocol such as NFS.

In one exemplary filer implementation, each filer 310, 312 can include a nonvolatile random access memory (NVRAM) 335 that provides fault-tolerant backup of data, enabling the integrity of filer transactions to survive a service interruption based upon a power failure, or other fault. The size of the NVRAM depends in part upon its implementation and function in the file server. It is typically sized sufficiently to log a certain time-based chunk of transactions (for example, several seconds worth). The NVRAM is filled, in parallel with the buffer cache, after each client request is completed, but before the result of the request is returned to the requesting client.

In an illustrative embodiment, the disks 360 are arranged into a plurality of volumes (for example, source volumes 314 and destination volumes 316), in which each volume has a file system associated therewith. The volumes each include one or more disks 360. In one embodiment, the physical disks 360 are configured into RAID groups so that some disks store striped data and some disks store separate parity for the data, in accordance with a preferred RAID 4 configuration. However, other configurations (e.g. RAID 5 having distributed parity across stripes) are also contemplated. In this embodiment, a minimum of one parity disk and one data disk is employed. However, a typical implementation may include three data and one parity disk per RAID group, and a multiplicity of RAID groups per volume.

B. Storage Operating System

To facilitate generalized access to the disks 360, the storage operating system 400 (Fig. 4) implements a write-anywhere file system that logically organizes the information as a hierarchical structure of directories and files on the disks. Each "on-disk" file may be implemented as a set of disk blocks configured to store information, such as data, whereas the directory may be implemented as a specially formatted file in which references to other files and directories are stored. As noted and defined above, in the illustrative embodiment described herein, the storage operating system is the NetApp® Data ONTAP™ operating system available from Network Appliance, Inc., of Sunnyvale, CA that implements the Write Anywhere File Layout (WAFL™) file system. It is expressly contemplated that any appropriate file system can be used, and as such, where the term "WAFL" is employed, it should be taken broadly to refer to any file system that is otherwise adaptable to the teachings of this invention.

The organization of the preferred storage operating system for each of the exemplary filers is now described briefly. However, it is expressly contemplated that the principles of this invention can be implemented using a variety of alternate storage operating system architectures. In addition, the particular functions implemented on each of the source and destination filers 310, 312 may vary. As shown in Fig. 4, the exemplary storage operating system 400 comprises a series of software layers, including a media access layer 405 of network drivers (e.g., an Ethernet driver). The operating system further includes network protocol layers, such as the Internet Protocol (IP) layer 410 and its supporting transport mechanisms, the Transport Control Protocol (TCP) layer 415 and the User Datagram Protocol (UDP) layer 420. A file system protocol layer provides multiprotocol data access and, to that end, includes support for the CIFS protocol 425, the NFS protocol 430 and the Hypertext Transfer Protocol (HTTP) protocol 435. In addition, the storage operating system 400 includes a disk storage layer 440 that implements a disk storage protocol, such as a RAID protocol, and a disk driver layer 445, that implements a disk control protocol such as the small computer system interface (SCSI).

Bridging the disk software layers with the network and file system protocol layers is a file system layer 450 of the storage operating system 400. Generally, the layer 450

implements a file system having an on-disk format representation that is block-based using, e.g., 4-kilobyte (KB) data blocks and using inodes to describe the files. In response to transaction requests, the file system generates operations to load (retrieve) the requested data from volumes if it is not resident "in-core", i.e., in the filer's memory 325. If the information is not in memory, the file system layer 450 indexes into the inode file using the inode number to access an appropriate entry and retrieve a volume block number. The file system layer 450 then passes the volume block number to the disk storage (RAID) layer 440, which maps that volume block number to a disk block number and sends the latter to an appropriate driver (for example, an encapsulation of SCSI implemented on a fibre channel disk interconnection) of the disk driver layer 445. The disk driver accesses the disk block number from volumes and loads the requested data in memory 325 for processing by the filer 310, 312. Upon completion of the request, the filer (and storage operating system) returns a reply, e.g., a conventional acknowledgement packet 374 defined by the CIFS specification, to the client 370 over the respective network connection 372.

It should be noted that the software "path" 470 through the storage operating system layers described above needed to perform data storage access for the client request received at the filer may alternatively be implemented in hardware or a combination of hardware and software. That is, in an alternate embodiment of the invention, the storage access request data path 470 may be implemented as logic circuitry embodied within a field programmable gate array (FPGA) or an application specific integrated circuit (ASIC). This type of hardware implementation increases the performance of the file service provided by filer 310, 312 in response to a file system request packet 374 issued by the client 370.

Overlying the file system layer 450 is the snapshot mirroring (or replication) application 490 in accordance with an illustrative embodiment of this invention. This application, as described in detail below, is responsible (on the source side) for the scanning and transmission of changes in the snapshot from the source filer 310 to the destination filer 312 over the network. This application is responsible (on the destination side) for the generation of the updated mirror snapshot from received information. Hence, the particular function of the source and destination applications are different, and are de-

scribed as such below. The snapshot mirroring application 490 operates outside of the normal request path 470 as shown by the direct links 492 and 494 to the TCP/IP layers 415, 410 and the file system snapshot mechanism (480). Notably, the application interacts with the file system layer to gain knowledge of files so it is able to use a file-based data structure (inode files, in particular) to replicate source snapshots at the destination.

C. Snapshot Procedures

The inherent Snapshot™ capabilities of the exemplary WAFL file system are further described in *TR3002 File System Design for an NFS File Server Appliance* by David Hitz *et al.*, published by Network Appliance, Inc., which is hereby incorporated by reference. Note, "Snapshot" is a trademark of Network Appliance, Inc. It is used for purposes of this patent to designate a persistent consistency point (CP) image. A persistent consistency point image (PCPI) is a point-in-time representation of the storage system, and more particularly, of the active file system, stored on a storage device (e.g., on disk) or in other persistent memory and having a name or other unique identifiers that distinguishes it from other PCPIs taken at other points in time. A PCPI can also include other information (metadata) about the active file system at the particular point in time for which the image is taken. The terms "PCPI" and "snapshot" shall be used interchangeably through out this patent without derogation of Network Appliance's trademark rights.

Snapshots are generally created on some regular schedule. This schedule is subject to great variation. In addition, the number of snapshots retained by the filer is highly variable. Under one storage scheme, a number of recent snapshots are stored in succession (for example, a few days worth of snapshots each taken at four-hour intervals), and a number of older snapshots are retained at increasing time spacings (for example, a number of daily snapshots for the previous week(s) and weekly snapshot for the previous few months). The snapshot is stored on-disk along with the active file system, and is called into the buffer cache of the filer memory as requested by the storage operating system 400 or snapshot mirror application 490 as described further below. However, it is contemplated that a variety of snapshot creation techniques and timing schemes can be implemented within the teachings of this invention.

An exemplary file system inode structure 500 according to an illustrative embodiment is shown in Fig. 5. The *inode for the inode file* or more generally, the "root" inode 505 contains information describing the inode file 508 associated with a given file system. In this exemplary file system inode structure root inode 505 contains a pointer to the inode file indirect block 510. The inode file indirect block 510 points to one or more inode file direct blocks 512, each containing a set of pointers to inodes 515 that make up the inode file 508. The depicted subject inode file 508 is organized into volume blocks (not separately shown) made up of inodes 515 which, in turn, contain pointers to file data (or "disk") blocks 520A, 520B and 520C. In the diagram, this is simplified to show just the inode itself containing pointers to the file data blocks. Each of the file data blocks 520(A-C) is adapted to store, in the illustrative embodiment, 4 kilobytes (KB) of data. Note, however, where more than a predetermined number of file data blocks are referenced by an inode (515) one or more indirect blocks 525 (shown in phantom) are used. These indirect blocks point to associated file data blocks (not shown). If an inode (515) points to an indirect block, it cannot also point to a file data block, and *vice versa*.

When the file system generates a snapshot of a given file system, a snapshot inode is generated as shown in Fig. 6. The snapshot inode 605 is, in essence, a duplicate copy of the root inode 505 of the file system 500. Thus, the exemplary file system structure 600 includes the same inode file indirect block 510, inode file direct block 512, inodes 515 and file data blocks 520(A-C) as depicted in Fig. 5. When a user modifies a file data block, the file system layer writes the new data block to disk and changes the active file system to point to the newly created block. The file layer does not write new data to blocks which are contained in snapshots.

Fig. 7 shows an exemplary inode file system structure 700 after a file data block has been modified. In this illustrative example, file data which is stored at disk block 520C is modified. The exemplary WAFL file system writes the modified contents to disk block 520C', which is a new location on disk. Because of this new location, the inode file data which is stored at disk block (515) is rewritten so that it points to block 520C'. This modification causes WAFL to allocate a new disk block (715) for the updated version of the data at 515. Similarly, the inode file indirect block 510 is rewritten to block 710 and

direct block 512 is rewritten to block 712, to point to the newly revised inode 715. Thus, after a file data block has been modified the snapshot inode 605 contains a pointer to the original inode file system indirect block 510 which, in turn, contains a link to the inode 515. This inode 515 contains pointers to the original file data blocks 520A, 520B and 520C. However, the newly written inode 715 includes pointers to unmodified file data blocks 520A and 520B. The inode 715 also contains a pointer to the modified file data block 520C' representing the new arrangement of the active file system. A new file system root inode 705 is established representing the new structure 700. Note that metadata in any snapshotted blocks (e.g. blocks 510, 515 and 520C) protects these blocks from being recycled or overwritten until they are released from all snapshots. Thus, while the active file system root 705 points to new blocks 710, 712, 715 and 520C', the old blocks 510, 515 and 520C are retained until the snapshot is fully released.

In accordance with an illustrative embodiment of this invention the source utilizes two snapshots, a "base" snapshot, which represents the image of the replica file system on the destination, and an "incremental" snapshot, which is the image that the source system intends to replicate to the destination, to perform needed updates of the remote snapshot mirror to the destination. In one example, from the standpoint of the source, the incremental snapshot can comprise a most-recent snapshot and the base can comprise a less-recent snapshot, enabling an up-to-date set of changes to be presented to the destination. This procedure shall now be described in greater detail.

D. Remote Mirroring

Having described the general procedure for deriving a snapshot, the mirroring of snapshot information from the source filer 310 (Fig. 3) to a remote destination filer 312 is described in further detail. As discussed generally above, the transmission of incremental changes in snapshot data based upon a comparison of changed blocks in the whole volume is advantageous in that it transfers only incremental changes in data rather than a complete file system snapshot, thereby allowing updates to be smaller and faster. However, a more efficient and/or versatile procedure for incremental remote update of a destination mirror snapshot is contemplated according to an illustrative embodiment of this

invention. Note, as used herein the term "replica snapshot," "replicated snapshot" or "mirror snapshot" shall be taken to also refer generally to the file system on the destination volume that contains the snapshot where appropriate (for example where a *snapshot of a snapshot* is implied.

As indicated above, it is contemplated that this procedure can take advantage of a sub-organization of a volume known as a qtree. A qtree acts similarly to limits enforced on collections of data by the size of a partition in a traditional Unix® or Windows® file system, but with the flexibility to subsequently change the limit, since qtrees have no connection to a specific range of blocks on a disk. Unlike volumes, which are mapped to particular collections of disks (e.g. RAID groups of *n* disks) and act more like traditional partitions, a qtree is implemented at a higher level than volumes and can, thus, offer more flexibility. Qtrees are basically an abstraction in the software of the storage operating system. Each volume may, in fact, contain multiple qtrees. The granularity of a qtree can be a sized to just as a few kilobytes of storage. Qtree structures can be defined by an appropriate file system administrator or user with proper permission to set such limits.

Note that the above-described qtree organization is exemplary and the principles herein can be applied to a variety of file system organizations including a whole-volume approach. A qtree is a convenient organization according to the illustrative embodiment, at least in part, because of its available identifier in the inode file.

Before describing further the process of deriving changes in two source snapshots, from which data is transferred to a destination for replication of the source at the destination, general reference is made again to the file block structures shown in Figs. 5-7. Every data block in a file is mapped to disk block (or volume block). Every disk/volume block is enumerated uniquely with a discrete volume block number (VBN). Each file is represented by a single inode, which contains pointers to these data blocks. These pointers are VBNs—each pointer field in an inode having a VBN in it, whereby a file's data is accessed by loading up the appropriate disk/volume block with a request to the file system (or disk control) layer. When a file's data is altered, a new disk block is allocated to store the changed data. The VBN of this disk block is placed in the pointer field of the inode. A snapshot captures the inode at a point in time, and all the VBN fields in it.

In order to scale beyond the maximum number of VBN "pointers" in an inode, "indirect blocks" are used. In essence, a disk block is allocated and filled with the VBNs of the data blocks, the inode pointers then point to the indirect block. There can exist several levels of indirect blocks, which can create a large tree structure. Indirect blocks are modified in the same manner as regular data blocks are—every time a VBN in an indirect block changes, a new disk/volume block is allocated for the altered data of the indirect block.

1. Source

Fig. 8 shows an exemplary pair of snapshot inode files within the source environment 800. In an illustrative embodiment, these represent two snapshots' inode files: the base 810 and incremental 812. Note that these two snapshots were taken at two points in time; the base represents the current image of the replica, and the incremental represents the image the replica will be updated to. The differences between the two snapshots define which changes are to be derived and committed to the remote replica/mirror. The inode files may each be loaded into the buffer cache of the source file server memory from the on-disk versions thereof using conventional disk access processes as directed by the storage operating system snapshot manager (480 in Fig. 4). In one embodiment, the base and incremental snapshots are loaded in increments as they are worked on by the operating system (rather than all-at-once). Each snapshot inode file 810, 812 is organized into a series of storage blocks 814. In this illustrative example, the base snapshot inode file 810 contains storage blocks denoted by volume (disk) block numbers, 5, 6 and 7, while the incremental snapshot inode file contains exemplary storage blocks having volume block numbers 3, 5, 6 and 8. Within each of the blocks are organized a given number of inodes 816. The volume blocks are indexed in the depicted order based upon their underlying logical file block placement.

In the example of a write-anywhere file layout, storage blocks are not immediately overwritten or reused. Thus changes in a file comprised of a series of volume blocks will always result in the presence of a new volume block number (newly written-to) that can be detected at the appropriate logical file block offset relative to an old block.

The existence of a changed volume block number at a given offset in the index between the base snapshot inode file and incremental snapshot inode file generally indicates that one or more of the underlying inodes and files to which the inodes point have been changed. Note, however, that the system may rely on other indicators of changes in the inodes or pointers—this may be desirable where a write-in-place file system is implemented.

A scanner 820 searches the index for changed base/incremental inode file snapshot blocks, comparing volume block numbers or another identifier. In the example of Fig. 8, block 4 in the base snapshot inode file 810 now corresponds in the file scan order to block 3 in the incremental snapshot inode file 812. This indicates a change of one or more underlying inodes. In addition, block 7 in the base snapshot inode file appears as block 8 in the incremental snapshot inode file. Blocks 5 and 6 are unchanged in both files, and thus, are quickly scanned over without further processing of any inodes or other information. Hence, scanned blocks at the same index in both snapshots can be efficiently bypassed, reducing the scan time.

Block pairs (e.g. blocks 7 and 8) that have been identified as changed are forwarded (as they are detected by the scan/scanner 820) to the rest of the source process, which includes an inode picker process 830. The inode picker identifies specific inodes (based upon qtree ID) from the forwarded blocks that are part of the selected qtree being mirrored. In this example the qtree ID Q2 is selected, and inodes containing this value in their file metadata are "picked" for further processing. Other inodes not part of the selected qtree(s) (e.g. inodes with qtree IDs Q1 and Q3) are discarded or otherwise ignored by the picker process 830. Note that a multiplicity of qtree IDs can be selected, causing the picker to draw out a group of inodes—each having one of the selected qtree associations.

The appropriately "picked" inodes from changed blocks are then formed into a running list or queue 840 of changed inodes 842. These inodes are denoted by a discrete inode number as shown. Each inode in the queue 840 is handed off to an inode handler or worker 850, 852 and 854 as a worker becomes available. Fig. 8A is a table 835 de-

tailing the basic set of rules the inode picker process 830 uses to determine whether to send a given inode to the queue for the workers to process.

The inode picker process 830 queries whether either (1) the base snapshot's version of the subject inode (a given inode number) is allocated *and* in a selected qtree (box 860) or (2) the incremental snapshot's version of the inode is allocated *and* in a selected qtree (box 862). If neither the base nor incremental version are allocated and in the selected qtree then both inodes are ignored (box 864) and the next pair of inode versions are queried.

If the base inode is not in allocated or not in the selected qtree, but the incremental inode is allocated and in the selected qtree, then this implies an incremental file has been added, and the appropriate inode change is sent to the workers (box 866). Similarly, if the base inode is allocated and in the selected qtree, but the incremental inode is not allocated or not in the selected qtree, then the this indicates a base file has been deleted and this is sent on to the destination via the data stream format (as described below) (box 868).

Finally, if a base inode and incremental inode are both allocated and in the selected qtree, then the process queries whether the base and incremental inodes represent the same file (box 870). If they represent the same file, then the file or its metadata (permissions, owner, permissions, etc) *may* have changed. This is denoted by different generation numbers on different versions of the inode number being examined by the picker process. In this case, a *modified* file is sent and the inode workes compare versions to determine exact changes as described further below (box 872). If the base and incremental are not the exact same file, then this implies a deletion of the base file and addition of an incremental file (box 874). The addition of the incremental file is noted as such by the picker in the worker queue.

Fig. 8B is a more detailed view of the information contained in exemplary changed blocks (block 10) in the base snapshot 810 and (block 12) in the incremental snapshot 812, respectively. Inode 2800 is unallocated in the base inode file and allocated in the incremental inode file. This implies that the file has been added to the file system. The inode picker process also notes that this inode is in the proper qtree Q2 (in this ex-

ample). This inode is sent to the changed inode queue for processing, with a note that the whole file is new.

Inode 2801 is allocated in both inode files. It is in the proper qtree Q2, and the two versions of this inode share the same generation number. This means that the inode represents the same file in the base and the incremental snapshots. It is unknown at this point whether the file data itself has changed, so the inode picker sends the pair to the changed inode queue, and a worker determines what data has changed. Inode 2802 is allocated in the base inode file, but not allocated in the incremental inode file. The base version of the inode was in the proper qtree Q2. This means this inode has been deleted. The inode picker sends this information down to the workers as well. Finally, inode 2803 is allocated in the base inode file, and *reallocated* in the incremental inode file. The inode picker 830 can determine this because the generation number has changed between the two versions (from #1 - #2). The new file which this inode represents has been added to the qtree, so like inode 2800, this is sent to the changed inode queue for processing, with a note that the whole file is new.

A predetermined number of workers operate on the queue 840 at a given time. In the illustrative embodiment, the workers function in parallel on a group of inodes in the queue. That is, the workers process inodes to completion in no particular order once taken from the queue and are free process further inodes from the queue as soon as they are available. Other processes, such as the scan 820 and picker 830 are also interleaved within the overall order.

The function of the worker is to determine changes between each snapshot's versions of the files and directories. As described above, the source snapshot mirror application is adapted to analyze two versions of inodes in the two snapshots and compares the pointers in the inodes. If the two versions of the pointers point to the same block, we know that that block hasn't changed. By extension, if the pointer to an indirect block has not changed, then that indirect block has no changed data, so none of its pointers can have changed, and, thus, *none of the data blocks underneath it* in the tree have changed. This means that, in a very large file, which is mostly unchanged between two snapshots,

the process can skip over/overlook VBN "pointers" to each data block in the tree to query whether the VBNs of the data blocks have changed.

The operation of a worker 850 is shown by way of example in Fig. 9. Once a changed inode pair are received by the worker 850, each inode (base and incremental, respectively) 910 and 912 is scanned to determine whether the file offset between respective blocks is a match. In this example, blocks 6 and 7 do not match. The scan then continues down the "tree" of blocks 6 and 7, respectively, arriving at underlying indirect blocks 8/9 (920) and 8/10 (922). Again the file offset comparison indicates that blocks 8 both arrive at a common block 930 (and thus have not changed). Conversely, blocks 9 and 10 do not match due to offset differences and point to changed blocks 940 and 942. The changed block 942 and the metadata above can be singled out for transmission to the replicated snapshot on the destination (described below; see also Fig. 8). The tree, in an illustrative embodiment extends four levels in depth, but this procedure may be applied to any number of levels. In addition, the tree may in fact contain several changed branches, requiring the worker (in fact, the above-described scanner 820 process) to traverse each of the branches in a recursive manner until all changes are identified. Each inode worker, thus provides the changes to the network for transmission in a manner also described below. In particular, new blocks and information about old, deleted blocks are sent to the destination. Likewise, information about modified blocks is sent.

Notably, because nearly every data structure in this example is a file, the above-described process can be applied not only to file data, but also to directories, access control lists (ACLs) and the inode file itself.

It should be again noted, that the source procedure can be applied to any level of granularity of file system organization, including an entire volume inode file. By using the inherent tree organization a quick and effective way to replicate a known subset of the volume is provided.

2. Communication Between Source and Destination

With further reference to Fig. 10, the transmission of changes from the source snapshot to the replicated destination snapshot is described in an overview 1000. As already described, the old and new snapshots present the inode picker 830 with changed inodes corresponding to the qtree or other selected sub-organization of the subject volume. The changed inodes are placed in the queue 840, and then their respective trees are walked for changes by a set of inode workers 850, 852 and 854. The inode workers each send messages 1002, 1004 and 1006 containing the change information to a source pipeline 1010. Note that this pipeline is only an example of a way to implement a mechanism for packaging file system data into a data stream and sending that stream to a network layer. The messages are routed first to a receiver 1012 that collects the messages and sends them on to an assembler 1014 as a group comprising the snapshot change information to be transmitted over the network 318. Again, the "network" as described herein should be taken broadly to include anything that facilitates transmission of volume sub-organization (e.g. qtree) change data from a source sub-organization to a destination sub-organization, even where source and destination are on the same file server, volume or, indeed (in the case of rollback as described in the above-incorporated U.S. Patent Application entitled SYSTEM AND METHOD FOR REMOTE ASYNCHRONOUS MIRRORING USING SNAPSHOTS) are the same sub-organization at different points in time. An example of a "network" used as a path back to the same volume is a loopback. The assembler 1014 generates a specialized format 1020 for transmitting the data stream of information over the network 318 that is predictable and understood by the destination. The networker 1016 takes the assembled data stream and forwards it to a networking layer. This format is typically encapsulated within a reliable networking protocol such as TCP/IP. Encapsulation can be performed by the networking layer, which constructs, for example, TCP/IP packets of the formatted replication data stream

The format 1020 is described further below. In general, its use is predicated upon having a structure that supports multiple protocol attributes (e.g. Unix permissions, NT access control lists (ACLs), multiple file names, NT streams, file type, file-create/modify time, etc.). The format should also identify the data in the stream (i.e. the offset location

in a file of specific data or whether files have "holes" in the file offset that should remain free). The names of files should also be relayed by the format. More generally, the format should also be independent of the underlying network protocol or device (in the case of a tape or local disk/non-volatile storage) protocol and file system—that is, the information is system "agnostic," and not bound to a particular operating system software, thereby allowing source and destination systems of different vendors to share the information. The format should, thus, be completely self-describing requiring no information outside the data stream. In this manner a source file directory of a first type can be readily translated into destination file directory of a different type. It should also allow extensibility, in that newer improvements to the source or destination operating system should not affect the compatibility of older versions. In particular, a data set (e.g. a new header) that is not recognized by the operating system should be ignored or dealt with in a predictable manner without triggering a system crash or other unwanted system failure (i.e. the stream is backwards compatible). This format should also enable transmission of a description of the whole file system, or a description of only changed blocks/information within any file or directory. In addition, the format should generally minimize network and processor overhead.

As changed information is forwarded over the network, it is received at the destination pipeline piece 1030. This pipeline also includes a networker 1032 to read out TCP/IP packets from the network into the snapshot replication data stream format 1020 encapsulated in TCP/IP. A data reader and header stripper 1034 recognizes and responds to the incoming format 1020 by acting upon information contained in various format headers (described below). A file writer 1036 is responsible for placing file data derived from the format into appropriate locations on the destination file system.

The destination pipeline 1030 forwards data and directory information to the main destination snapshot mirror process 1040, which is described in detail below. The destination snapshot mirror process 1040 consists of a directory stage 1042, which builds the new replicated file system directory hierarchy on the destination side based upon the received snapshot changes. To briefly summarize, the directory stage creates, removes and moves files based upon the received formatted information. A map of inodes from the

destination to the source is generated and updated. In this manner, inode numbers on the source file system are associated with corresponding (but typically different) inode numbers on the destination file system. Notably, a temporary or "purgatory" directory 1050 (described in further detail below) is established to retain any modified or deleted directory entries 1052 until these entries are reused by or removed from the replicated snapshot at the appropriate directory rebuilding stage within the directory stage. In addition, a file stage 1044 of the destination mirror process populates the established files in the directory stage with data based upon information stripped from associated format headers.

The format into which source snapshot changes are organized is shown schematically in Figs. 11 and 12. In the illustrative embodiment, the format is organized around 4 KB blocks. The header size and arrangement can be widely varied in alternate embodiments, however. There are 4 KB headers (1100 in Fig. 11) that are identified by certain "header types." Basic data stream headers ("data") are provided for at most every 2 megabytes (2 MB) of *changed* data. With reference to Fig. 11, the 4 KB standalone header includes three parts, a 1 KB generic part 1102, a 2 KB non-generic part 1104, and an 1 KB expansion part. The expansion part is not used, but is available for later versions.

The generic part 1102 contains an identifier of header type 1110. Standalone header types (i.e. headers not followed by associated data) can indicate a start of the data stream; an end of part one of the data stream; an end of the data stream; a list of deleted files encapsulated in the header; or the relationship of any NT *streamdirs*. Later versions of Windows NT allow for multiple NT "streams" related to particular filenames. A discussion of streams is found in U.S. Patent Application Serial No. 09/891,195, entitled SYSTEM AND METHOD FOR REPRESENTING NAMED DATA STREAMS WITHIN AN ON-DISK STRUCTURE OF A FILE SYSTEM, by Kayuri Patel, *et al*, the teachings of which are expressly incorporated herein by reference. Also in the generic part 1102 is a checksum 1112 that ensures the header is not corrupted. In addition other data such as a "checkpoint" 1114 used by the source and destination to track the progress of replication is provided. By providing a list of header types, the destination can more easily operate in a backwards-compatible mode—that is, a header type that is not recog-

nized by the destination (provided from a newer version of the source) can be more easily ignored, while recognized headers within the limits of the destination version are processed as usual.

The kind of data in the non-generic part 1104 of the header 1100 depends on the header type. It could include information relating to file offsets (1120) in the case of the basic header, used for follow-on data transmission, deleted files (in a standalone header listing of such files that are no longer in use on the source or whose generation number has changed) (1122), or other header-specific information (1124 to be described below). Again, the various standalone headers are interposed within the data stream format at an appropriate location. Each header is arranged to either reference an included data set (such as deleted files) or follow-on information (such as file data).

Fig. 12 describes the format 1020 of the illustrative replication data stream in further detail. The format of the replicated data stream is headed by a standalone data stream header 1202 of the type "start of data stream." This header contains data in the non-generic part 1104 generated by the source describing the attributes of the data stream.

Next a series of headers and follow-on data in the format 1020 define various "part 1" information (1204). Significantly, each directory data set being transmitted is preceded by a basic header with no non-generic data. Only directories that have been modified are transmitted, and they need not arrive in a particular order. Note also that the data from any particular directory need not be contiguous. Each directory entry is loaded into a 4 KB block. Any overflow is loaded into a new 4 KB block. Each directory entry is a header followed by one or more names. The entry describes an inode and the directory names to follow. NT stream directories are also transmitted.

The part 1 format information 1204 also provides ACL information for every file that has an associated ACL. By transmitting the ACLs before their associated file data, the destination can set ACLs before file data is written. ACLs are transmitted in a "regular" file format. Deleted file information (described above) is sent with such information included in the non-generic part 1104 of one or more standalone headers (if any). By

sending this information in advance, the directory tree builder can differentiate between moves and deletes.

The part 1 format information 1204 also carries NT stream directory (streamdir) relationship information. One or more standalone headers (if any) notifies the destination file server of every changed file or directory that implicates NT streams, regardless of whether the streams have changed. This information is included in the non-generic part 1104 of the header 1100 (Fig. 11).

Finally, the part 1 format information 1204 includes special files for every change in a symlink, named pipe, socket, block device, or character device in the replicated data stream. These files are sent first, because they are needed to assist the destination in building the infrastructure for creation of the replicated file system before it is populated with file data. Special files are, like ACLs, transmitted in the format of regular files.

Once various part 1 information 1204 is transmitted, the format calls for an "end of part 1 of the data stream" header 1206. This is a basic header having no data in the non-generic part 1104. This header tells the destination that part 1 is complete and to now expect file data.

After the part 1 information, the format presents the file and stream data 1208. A basic header 1210 for every 2 MB or less of *changed* data in a file is provided, followed by the file data 1212 itself. The files comprising the data need not be written in a particular order, nor must the data be contiguous. In addition, referring to the header in Fig. 11, the basic header includes a block numbers data structure 1130, associated with the non-generic part 1104 works in conjunction with the "holes array" 1132 within (in this example) the generic part 1102. The holes array denotes empty space. This structure, in essence, provides the mapping from the holes array to corresponding blocks in the file. This structure instructs the destination where to write data blocks or holes.

In general files (1212) are written in 4 KB chunks with basic headers at every 512 chunks (2 MB), at most. Likewise, streams (also 1212) are transmitted like regular files in 4 KB chunks with at most 2 MB between headers.

Finally, the end of the replicated data stream format 1020 is marked by a footer 1220 consisting of standalone header of the type "end of data stream." This header has no specific data in its non-generic part 1104 (Fig. 11).

3. Destination

When the remote destination (e.g. a remote file server, remote volume, remote qtree or the same qtree) receives the formatted data stream from the source file server via the network, it creates a new qtree or modifies an existing mirrored qtree (or another appropriate organizational structure) and fills it with data. Fig. 13 shows the destination snapshot mirror process 1040 in greater detail. As discussed briefly above, the process consists of two main parts, a directory stage 1042 and a data or file stage 1044.

The directory stage 1042 is invoked first during a transmission the data stream from the source. It consists of several distinct parts. These parts are designed to handle all part 1 format (non-file) data. In an illustrative embodiment the data of part 1 is read into the destination, stored as files locally, and then processed from local storage. However, the data may alternatively be processed as it arrives in realtime.

More particularly, the first part of the directory stage 1042 involves the processing of deleted file headers (1310). Entries in the inode map (described further below) are erased with respect to deleted files, thereby severing a relation between mapped inodes on the replicated destination snapshot and the source snapshot.

Next the directory stage undertakes a tree cleaning process (1312). This step removes all directory entries form the replicated snapshot directory 1330 that have been changed on the source snapshot. The data stream format (1020) indicates whether a directory entry has been added or removed. In fact, directory entries from the base version of the directory and directory entries from the incremental version of the directory are both present in the format. The destination snapshot mirror application converts the formatted data stream into a destination directory format in which each entry that includes an inode number, a list of relative names (e.g. various multi-protocol names) and a "create" or "delete" value. In general each file also has associated therewith a generation

number. The inode number and the generation number together form a *tuple* used to directly access a file within the file system (on both the source and the destination). The source sends this tuple information to the destination within the format and the appropriate tuple is stored on the destination system. Generation numbers that are out of date with respect to existing destination files indicate that the file has been deleted on the source. The use of generation numbers is described further below.

The destination processes base directory entries as removals and incremental directory entries as additions. A file which has been moved or renamed is processed as a delete (from the old directory or from the old name), then as an add (to the new directory or with a new name). Any directory entries 1952 that are deleted, or otherwise modified, are moved temporarily to the temporary or "purgatory" directory, and are not accessible in this location by users. The purgatory directory allows modified entries to be, in essence, "moved to the side" rather than completely removed as the active file system's directory tree is worked on. The purgatory directory entries, themselves point to data, and thus prevent the data from becoming deleted or losing a link to a directory altogether.

On a base transfer of a qtree to the destination, the directory stage tree building process is implemented as a breadth-first traversal of all the files and directories in the data stream, starting with the root of the qtree. The directory stage then undertakes the tree building process, which builds up all the directories with stub entries for the files. However, the depicted incremental directory stage (1042), as typically described herein, differs from a base transfer in that the tree building process (1314) begins with a directory queue that includes *all* modified directories currently existing on both the source and the destination (i.e. the modified directories that existed prior to the transfer). The incremental directory stage tree building process then processes the remainder of the directories according to the above-referenced breadth-first approach.

For efficiency, the source side depends upon inode numbers and directory blocks rather than pathnames. In general, a file in the replicated directory tree (a qtree in this example) on the destination cannot expect to receive the same inode number as the corresponding file has used on the source (although it is possible). As such, an *inode map* is

established in the destination. This map 1400, shown generally in Fig. 14, enables the source to relate each file on the source to the destination. The mapping is based generally upon file offsets. For example a received source block having "offset 20KB in inode 877" maps to the block at offset 20 KB in replicated destination inode 9912. The block can then be written to the appropriate offset in the destination file.

More specifically, each entry in the inode map 1400 contains an entry for each inode on the source snapshot. Each inode entry 1402 in the map is indexed and accessed via the source inode number (1404). These source inodes are listed in the map in a sequential and monotonically ascending order, notwithstanding the order of the mapped destination inodes. Under each source inode number (1404), the map includes: the source generation number (1406) to verify that the mapped inode matches the current file on the source; the destination inode number (1408); and destination generation number (1410). As noted above, the inode number and generation number together comprise a *tuple* needed to directly access an associated file in the corresponding file system.

By maintaining the source generation number, the destination can determine if a file has been modified or deleted on the source (and its source associated inode reallocated), as the source generation number is incremented upwardly with respect to the stored destination. When the source notifies the destination that an inode has been modified, it sends the tuple to the destination. This tuple uniquely identifies the inode on the source system. Each time the source indicates that an entirely new file or directory has to be created (e.g. "create") the destination file system creates that file. When the file is created, the destination registers data as a new entry in its inode map 1400. Each time the source indicates that an existing file or directory needs to be deleted, the destination obliterates that file, and then clears the entry in the inode map. Notably, when a file is modified, the source only sends the tuple and the data to be applied. The destination loads the source inode's entry from the inode map. If the source generation number matches, then it knows that the file already exists on the destination and needs to be modified. The destination uses the tuple recorded in the inode map to load the destination inode. Finally, it can apply the file modifications by using the inode.

As part of the tree building process reused entries are "moved" back from the purgatory directory to the replicated snapshot directory 1330. Traditionally, a move of a file requires knowledge of the name of the moved file and the name of the file it is being moved to. The original name of the moved file may not be easily available in the purgatory directory. In addition, a full move would require two directories (purgatory and replicated snapshot) to be modified implicating additional overhead.

However, in the illustrative embodiment, if the source inodes received at the destination refer to inodes in the inode map 1400, then the directory stage creates (on the current built-up snapshot directory 1339) a file entry having the desired file name. This name can be exactly the name derived from the source. A hard link 1332 (i.e. a Unix-based link enables multiple names to be assigned to a discrete file) is created between that file on the snapshot directory 1330 and the entry in the purgatory directory. By so linking the entry, it is now pointed to by both the purgatory directory and the file on the snapshot directory itself. When the purgatory directory root is eventually deleted (thereby killing off purgatory) at the end of the data stream transfer, the hard link will remain to the entry, ensuring that the specific entry in the purgatory directory will not be deleted or recycled (given that the entry's link count is still greater than zero) and a path to the data from the file on the new directory is maintained. Every purgatory entry that eventually becomes associated with a file in the newly built tree will be similarly hard linked, and thereby survive deletion of the purgatory directory. Conversely, purgatory entries that are not relinked will not survive, and are effectively deleted permanently when purgatory is deleted.

It should now be clear that the use of mapping and generation number tuples avoids the expensive (from a processing standpoint) use of conventional full file pathnames (or relative pathnames) in the data stream from the source. Files that are modified on the source can be updated on the destination without loading a directory on either the source or destination. This limits the information needed from the source and the amount of processing required. In addition, the source need not maintain a log of directory operations. Likewise, since the destination need not maintain a central repository of the current file system state, multiple subdirectories can be operated upon concurrently. Fi-

nally, neither the source, nor the destination must explicitly track deleted files as such deleted files are automatically removed. Rather, the source only sends its list of deleted files and the destination uses this list to conform the inode map. As such, there is no need to selectively traverse a tree more than once to delete files, and at the conclusion of the transfer, simply eliminating the purgatory directory is the only specific file cleaning step.

The directory stage 1042 sets up any ACLs on directories as the directories are processed during tree building (substep 1316). As described above, the ACL and NT stream relationships to files are contained in appropriate standalone headers. ACLs are then set on files during the below-described file stage. NT streams are created on files as the files are, themselves, created. Since an NT steam is, in fact, a directory, the entries for it are processed as part of the directory phase.

The new directory tree may contain files with no data or old data. When the "end of part 1" format header is read, the destination mirror process 1040 enters the file stage 1044 in which snapshot data files 1340 referenced by the directory tree are populated with data (e.g. change data). Fig. 15 shows a simplified procedure 1500 for writing file data 1502 received from the source. In general, each (up to) 2 MB of data in 4 KB blocks arrives with corresponding source inode numbers. The inode map 1400 is consulted for corresponding entries 1402. Appropriate offsets 1504 are derived for the data, and it is written into predetermined empty destination snapshot data files 1340.

At the end of both the directory stage 1042 and data stage 1044, when all directory and file data have been processed, and the data stream transfer from the source is complete, the new replicated snapshot is exposed atomically to the user. At this time the contents of the purgatory directory 1050 (which includes any entries that have not be "moved" back into the rebuilt tree) is deleted.

It should be noted that the initial creation (the "level zero" transfer) of the replicated snapshot on the destination follows the general procedures discussed above. The difference between a level zero transfer and a regular update is that there is no base snapshot; so the comparisons always process information in the incremental snapshot as additions and creates rather than modifications. The destination mirror application starts tree

building by processing any directories already known to it. The initial directory established in the destination is simply the root directory of the replicated snapshot (the qtree root). A destination root exists on the inode map. The source eventually transmits a root (other files received may be buffered until the root arrives), and the root is mapped to the existing destination root. Files referenced in the root are then mapped in turn in a "create" process as they are received and read by the destination. Eventually, the entire directory is created, and then the data files are populated. After this, a replica file system is complete.

E. Rollback

As described above, a source and destination can be the same qtree, typically at different points in time. In this case, it is contemplated that an incremental change to a snapshot can be undone by applying a "rollback" procedure. In essence, the base and incremental snapshot update process described above with reference to Fig. 8 is performed in reverse so as to recover from a disaster, and return the active file system to the state of a given snapshot.

Reference is made to Fig. 16, which describes a generalized rollback procedure 1600 according to an illustrative embodiment. As a matter of ongoing operation, in step 1605, a "first" snapshot is created. This first snapshot may be an exported snapshot of the replicated snapshot on the destination. In the interim, the subject destination active file system (replicated snapshot) is modified by an incremental update from the source (step 1610).

In response to an exigency, such as a panic, crash, failure of the update to complete or a user-initiated command, a rollback initiation occurs (step 1615). This is a condition in which the next incremental update of the replicated snapshot will not occur properly, or otherwise does not reflect an accurate picture of the data.

In response to rollback initiation, further modification/update to the replicated snapshot is halted or frozen (step 1620). This avoids further modifications that may cause the active file system to diverge from the state to be reflected in a second snapshot

that will be created from the active file system in the next step (step 1625 below) immediately after the halt. Modification to the active file system is halted using a variety of techniques such as applying read only status to the file system or denying all access. In one embodiment, access to the active file system is redirected to an exported snapshot by introducing a level of indirection to the inode lookup of the active file system, as set forth in the above-incorporated U.S. Patent Application entitled SYSTEM AND METHOD FOR REDIRECTING ACCESS TO A REMOTE MIRRORED SNAPSHOT.

After the halt, a "second" exported snapshot of the modified active file system in its most current state is now created (step 1625).

Next, in step 1630, the incremental changes are computed between the second and the first snapshots. This occurs in accordance with the procedure described above with reference to Figs. 8 and 9, but using the second snapshot as the base and the first snapshot as the incremental. The computed incremental changes are then applied to the active file system (now frozen in its present state) in step 1635. The changes are applied so that the active file system is eventually "rolled back" to the state contained in the first snapshot (step 1640). This is the active file system state existing before the exigency that necessitated the rollback.

In certain situations, the halt or freeze on further modification of the active file system according to step 1625 is released, allowing the active file system to again be accessed for modification or user intervention (step 1645). However, in the case of certain processes, such as rollback (described below), a rolled back qtree is maintained under control for further modifications by the replication process.

One noted advantage to the rollback according to this embodiment is that it enables the undoing of set of changes to a replicated data set without the need to maintain separate logs or consuming significant system resources. Further the direction of rollback—past-to-present or present-to-past—is largely irrelevant. Furthermore, use of the purgatory directory, and not deleting files, enables the rollback to not affect existing NFS clients. Each NFS client accesses files by means of file handles, containing the inode number and generation of the file. If a system deletes and recreates a file, the file will

have a different inode/generation tuple. As such, the NFS client will not be able to access the file without reloading it (it will see a message about a stale file handle). The purgatory directory, however, allows a delay in unlinking files until the end of the transfer. As such, a rollback as described above can resurrect files that have just been moved into purgatory, without the NFS clients taking notice.

F. Inode Map Flip

Where a destination replicated snapshot may be needed at the source to, for example, rebuild the source qtree snapshot, (in other words, the role of the source and destination snapshot are reversed) the use of generalized rollback requires that the inode map be properly related between source and destination. This is because the source inodes do not match the destination inodes in their respective trees. For the same reason an inode map is used to construct the destination tree, the source must exploit a mapping to determine the nature of any inodes returned from the destination during the rollback. However, the inode map residing on the destination does not efficiently index the information in a form convenient for use by the source. Rather, the source would need to hunt randomly through the order presented in the map to obtain appropriate values.

One way to provide a source-centric inode map is to perform a "flip" of map entries. Fig. 17 details a procedure 1700 for performing the flip. The flip operation is initiated (step 1705) as part of a rollback initiated as part of a disaster recovery procedure of for other reasons (automatically or under user direction). Next, the destination and source negotiate to transfer the inode map file to the source from the destination. The negotiation can be accomplished using known data transfer methodologies and include appropriate error correction and acknowledgements (step 1710). The inode is thereby transferred to the source from the destination and is stored.

Next the source (which after the negotiation becomes the new destination), creates an empty inode map file with one entry for each inode in the source qtree (step 1715). The new destination then initializes a counter with (in this example) N=1 (step 1720). N is the variable representing the inode count on the new destination qtree.

In step 1725, the new destination looks up the Nth inode from the entries associated with the old destination in the stored inode map file (i.e. the map from the old destination/new source). Next, the new destination determines if such an entry exists (decision step 1730). If no entry exists, then a zero entry is created in the new inode map file, representing that the Nth inode of the new source (old destination) is not allocated. However, if there exists an Nth inode of the new source/old destination, then the decision step 1730 branches to step 1740, and creates a new entry in the new inode map file (created in step 1715). The new entry maps the new source (old destination) Nth inode to the proper new destination (old source) inode. Note, in an alternate embodiment, the new inode map is provided with a full field of zero entries before the mapping begins, and the creation of a "zero entry," in this case should be taken broadly to include leaving a preexisting zero entry in place in the inode map.

The procedure 1700 then checks if N equals the number of inodes in the old destination file system (decision step 1745). If so, the new inode map file is complete and the procedure quits (step 1750). Conversely, if additional inodes are still to-be-mapped, then the counter is incremented by one (N=N+1 in step 1755). Similarly, if a zero entry is made into the new inode map, then the procedure 1700 also branches to decision step 1745 to either increment the counter (step 1755) or quit (step 1750). Where the counter is incremented in step 1755, the procedure branches back to step 1725 wherein the incremented Nth inode is looked up.

By way of example, Fig. 18 shows an illustrative old destination inode map file 1800 including three exemplary entries 1802, 1804 and 1806, sequentially. The fields 1404, 1406 (source and destination inode numbers), 1408, 1410 (source and destination generation numbers) are described above with reference to Fig. 14. Entry 1802 shows that (old) source inode 72 maps to (old) destination inode 605. Likewise entry 1804 maps source inode 83 to destination inode 328, and entry 1806 maps source inode 190 to destination inode 150.

Fig 19 shows an exemplary new inode map file 1900 generated from the old inode map file 1800 of Fig. 18 in accordance with the flip procedure 1700. The new map in-

cludes fields for the new source (old destination) inode 1902, new destination (old source) inode 1904, new source (old destination) generation number 1906 and new destination (old source) generation number 1908. As a result of the flip, the entry 1910 for new source inode 150 is presented in appropriate index order and is paired with new destination inode 190 (and associated generation numbers). The entry 1912 for new source inode 328 is next (after a series of consecutive, intervening entries 1914 for new source inodes 151-372) and maps new destination inode 83. Likewise the entry 1916 for new source inode 605 maps new destination inode 72, after intervening entries 1918 for new source inodes 329-604. The intervening source inodes may contain mappings to other new existing destination inodes, or they may have a zero value as shown in entry 1930 for new source inode 606 (as provided by step 1735 of the procedure 1700 where no new destination inode was detected on the stored old source inode map (1800)).

G. Inode Map Association

It is further contemplated that, two replica/mirror snapshots of the same source can establish a mirror relationship with one another. These two snapshots may be representative of two different points in time with respect to the original source. Fig. 20 shows a generalized environment 2000 in which an original source 2001 has generated two replica/mirror snapshots Destination Snapshot A (2002) and Destination Snapshot B (2004). Each Destination Snapshot A and B (2002 and 2004) has an associated inode map A and B (2012 and 2014, respectively), used to map the inodes of transferred data stream from the original source 2001.

In the illustrated example, the Destination Snapshot A (2002) is now prepared to transfer changes so as to establish a mirror in Destination Snapshot B (2004). However, the reverse is also contemplated, i.e. Destination Snapshot B establishing a Mirror in Destination Snapshot A. Thus, Destination Snapshot A (2002) becomes the new "source" in the transfer with Destination Snapshot B (2004) acting as the desired destination system for replication data from Destination Snapshot A. As in the above-described flip embodiment, the new source 2002 transfers its inode map A 2012 to the destination system 2004. The destination system 2004 then determines the relationship between the

two system's inodes. In this case, both the new source and the new destination system have their own inode maps A and B (2012 and 2014), indexed off the old source 2001, and referencing the inodes in their respective trees. Given the existence of two respective inode maps, an "associative" process 2016 walks the inode maps concurrently, inode-by-inode. For each inode from the original source 2001, the process extracts the "destination inode/generation number" from each of the inode maps A and B. It then treats the new source as the appropriate map index for the new associated inode map 2018. In the associated map, it stores the new source generation number for the new source index inode number, with each index entry also associated with/mapped to the new destination inode/generation number extracted from the inode map B (2014). The new map is used by the new destination 2004 in accordance with the principles described above to build trees in the directory based upon changes in the new source with respect to various points in time.

By way of example, an hypothetical old source OS inode number 55 (OS 55) is mapped to old destination snapshot A in its map A to old destination A inode 87 (A 87) and OS 55 is mapped to old destination B inode 99 (B 99) in map B. To make B the new destination and A the new source, an associative map is constructed with the process extracting A 87 and B 99 for the respective maps based upon the common index OS 55. The associated map contains the new source/new destination entry 87/99. It also includes the associated generation numbers with these values from the old maps A and B. Note that, while the procedure is applied to two old destination systems, it is contemplated that more than two destination systems can be associated in various ways in accordance with the techniques described herein.

The foregoing has been a detail description of illustrative embodiments of the invention. Various modifications and additions can be made without departing from the spirit and scope of the invention. For example, the number of interconnected source and/or destination servers depicted can be varied. In fact, the source and destination servers can be the same machine. It is expressly contemplated that a plurality of sources can transfer data to a destination and *vice versa*. Likewise, the internal architecture of the servers or their respective storage arrays, as well as their network connectivity and proto-

cols, are all highly variable. The operating systems used on various source and destination servers can differ. In addition, it is expressly contemplated that any of the operations and procedures described herein can be implemented using hardware, software comprising a computer-readable medium having program instructions executing on a computer, or a combination of hardware and software.

What is claimed is:

1. A system for identifying changes in a logical group of data blocks on a source and updating a replica of the logical group of data blocks on a destination, comprising:

A) a scanner operative on a first set of block identifiers and a second set of block identifiers of a first snapshot and second snapshot respectively, the first snapshot and the second snapshot each respectively corresponding to a different image of the logical group of data blocks with the first snapshot corresponding to the replica, the scanner being adapted to search the second set of block identifiers for block identifiers that differ from corresponding block identifiers of the first set of block identifiers and thereby indicate changed data blocks that are not reflected in the replica;

B) transmitting the changed blocks to the destination without transmitting all the blocks of the logical group; and

C) updating the replica on the destination with the changed blocks.

2. The system of claim 1, wherein the scanner is adapted to walk down a hierarchy of pointers associated with respective logical block indexes for the first and second snapshots and identify pointed-to blocks with changed block identifiers relative to pointed-to blocks at corresponding offsets while bypassing blocks having unchanged volume block numbers.

3. The system of claim 1 further comprising an inode picker that picks out inodes in the retrieved blocks having a predetermined association and that indicate changes between, respectively a first version and a second version of the inodes in the first snapshot and the second snapshot.

4. The system of claim 3, wherein the predetermined association comprises a sub-organization of a volume file system.

5. The system of claim 4, wherein the sub-organization comprises a qtree within the volume file system.

6. The system of claim 4 further comprising an inode worker that operates on queued blocks retrieved by the inode picker and that applies the scanner to derive changes between blocks in a hierarchy of the first version of each of the picked out inodes with respect to a hierarchy of the second version of each of the picked out inodes.

7. The system of claim 6, wherein the inode worker is adapted to collect the changes an forward the changes to a pipeline that packages the changes in a data stream for transmission to the destination file system.

8. The system of claim 7 further comprising an extensible and file system-independent format that transmits the changes between the source file system and the destination file system.

9. A method for updating a replica of a logical group of data blocks on a destination with changes in a corresponding logical group on a source, comprising:

A) comparing a first set of block identifiers of a first snapshot of the logical group on the source, which corresponds to the replica, with a second set of block identifiers of a second snapshot of the logical group on the source to identify data blocks in corresponding locations within the logical group that have changed;

B) transmitting the changed blocks to the destination without transmitting all the blocks of the logical group; and

C) updating the replica on the destination with the transmitted changed blocks.

10. The method of claim 9, wherein the first snapshot and second snapshot correspond to images of the logical group on the source at respective first and second points in time, and the comparing step selects the corresponding block identifiers of the first and second set of block identifiers that have changed between the first and second points of time.

11. The method of claim 10, wherein the logical group on the source comprises a hierarchical arrangement, and the first and second sets of block identifiers each comprise

a hierarchical block index corresponding to the hierarchical arrangement, and the selecting step is performed by a scanner that searches and compares the hierarchical block indexes to identify the changed blocks.

12. The method of claim 11, wherein the block identifiers of the first and second sets of block identifiers refer to data blocks at corresponding locations within the logical group on the source, and the selecting step selects changed blocks by comparing block identifiers of the first and second sets of blocks identifiers that refer to data blocks at the same locations within the logical group.

13. The method of claim 9 further comprising picking out inodes, with an inode picker, in the retrieved blocks having a predetermined association and indicating changes between, respectively a first version and a second version of the inodes in the first snapshot and the second snapshot.

14. The method of claim 13, wherein the predetermined association comprises a sub-organization of a volume file system.

15. The method of claim 14, wherein the sub-organization comprises a qtree within the volume file system.

16. A system for receiving a data stream of changes from a snapshot of a logical group of data blocks on a source and updating a replica of the logical group on a destination, comprising:

A) a metadata stage process that reads metadata describing a structure for the logical group on the source and maps references to the logical group on the source with references to the logical group on the destination to identify changed blocks; and

B) a data stage process that, responsive to the mapped references, populates the logical group on the replica with data blocks from the source that have changed at corresponding offsets in the logical group.

17. A system for identifying changes to a logical group of data blocks on a source and updating a replica of the logical group on a destination, comprising:

A) a metadata stage process including the steps of (i) reading metadata describing a structure for the logical group on the source; the metadata including a first set of references to the data blocks of the logical group on the source; and (ii) generating a replica of the read metadata, including the step of mapping the first set of references to a second set of references to data blocks of the logical group on the destination; and

B) a data stage process responsive to the metadata stage process including the step of populating the replica with data blocks referenced by the second set of references.

18. A method for updating a replica on a destination, comprising:

A) reading, from changed data of the snapshot, identifiers related to deleted and modified logical groups of data on the replica and placing the deleted and modified logical groups in a temporary store separate from a main store of the replica;

B) creating a set of references in the main store to the deleted and modified logical groups in the temporary store; and

C) after the creating step, deallocating the temporary store while maintaining the references in the main store to the deleted and modified logical groups of data.

ABSTRACT OF THE DISCLOSURE

A system and method for remote asynchronous replication or mirroring of changes in a source file system snapshot in a destination replica file system using a scan (via a scanner) of the blocks that make up two versions of a snapshot of the source file system, which identifies changed blocks in the respective snapshot files based upon differences in volume block numbers identified in a scan of the logical file block index of each snapshot. Trees of blocks associated with the files are traversed, bypassing unchanged pointers between versions and walking down to identify the changes in the hierarchy of the tree. These changes are transmitted to the destination mirror or replicated snapshot. This technique allows regular files, directories, inodes and any other hierarchical structure to be efficiently scanned to determine differences between versions thereof. The changes in the files and directories are transmitted over the network for update of the replicated destination snapshot in an asynchronous (lazy write) manner. The changes are described in an extensible, system-independent data stream format layered under a network transport protocol. At the destination, source changes are used to update the destination snapshot. Any deleted or modified inodes already on the destination are moved to a temporary or "purgatory" directory and, if reused, are relinked to the rebuilt replicated snapshot directory. The source file system snapshots can be representative of a volume sub-organization, such as a qtree.

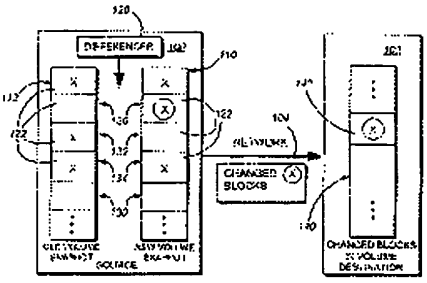


FIG. 1
(PRIOR ART)

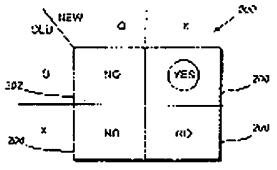


FIG. 2
(PRIOR ART)

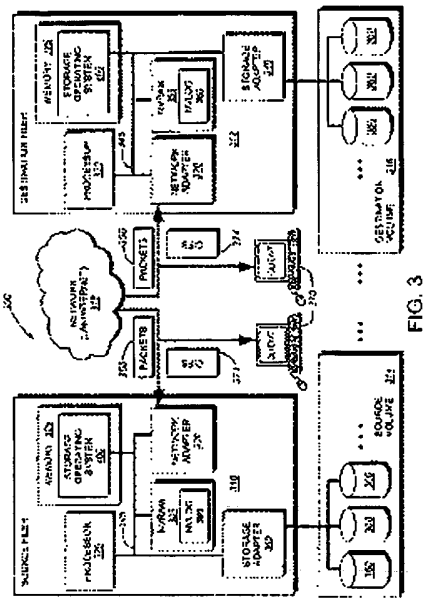


FIG. 3

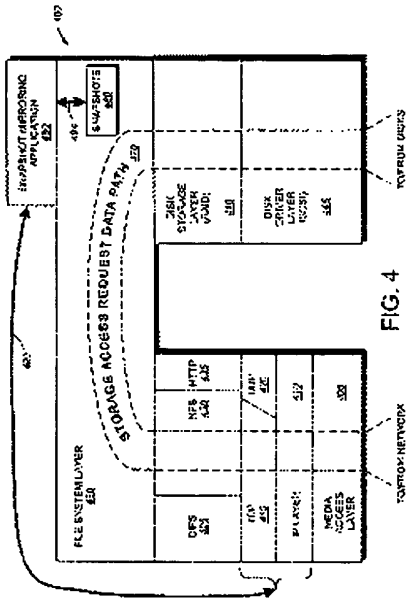


FIG. 4

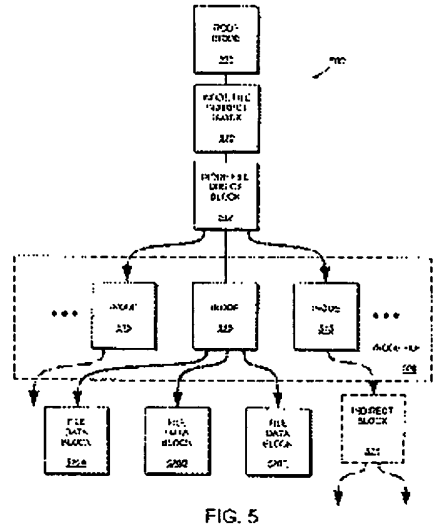


FIG. 5

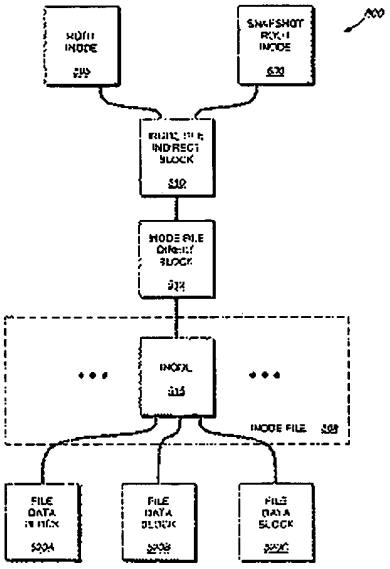


FIG. 6

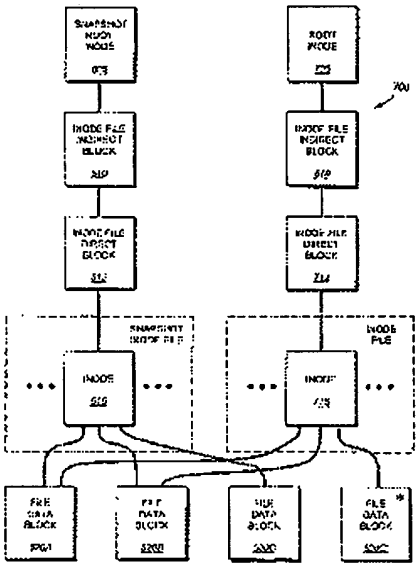


FIG. 7

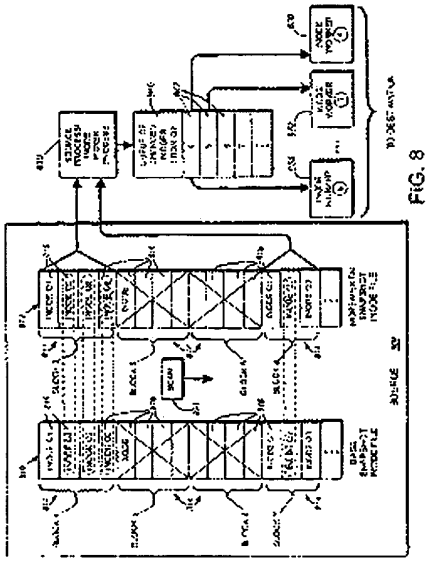


FIG. 8

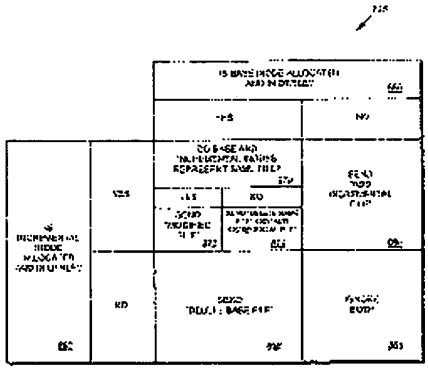


FIG. 8A

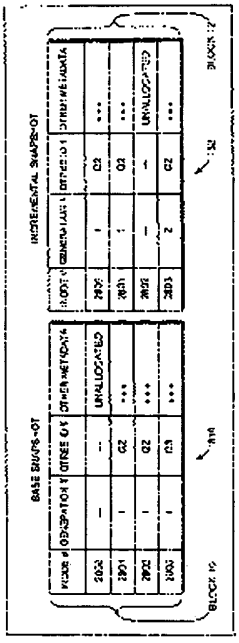


FIG. 8B

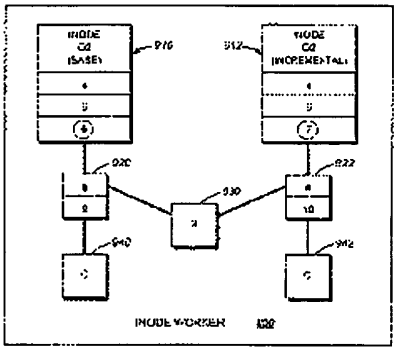


FIG. 9

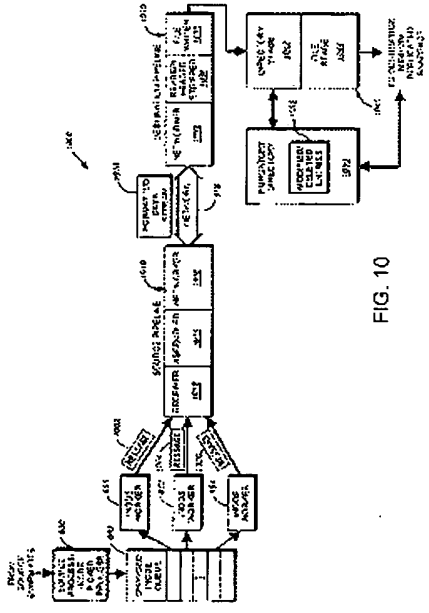


FIG. 10

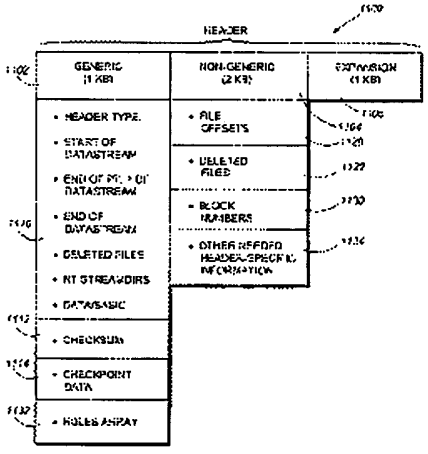


FIG. 11

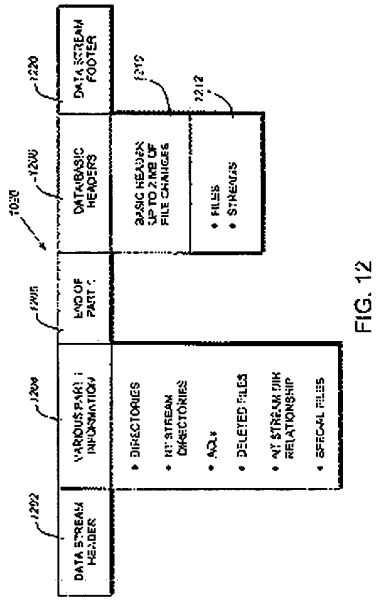


FIG. 12

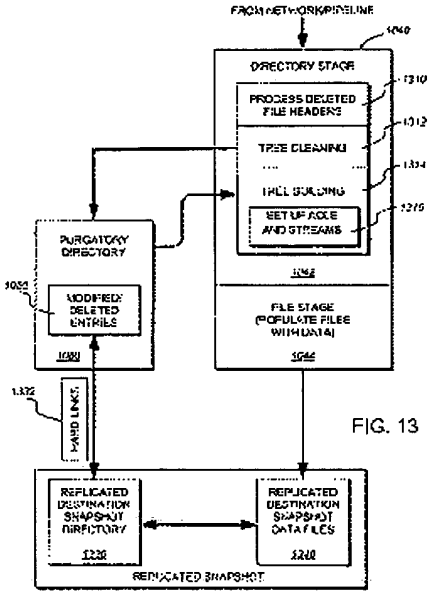


FIG. 13

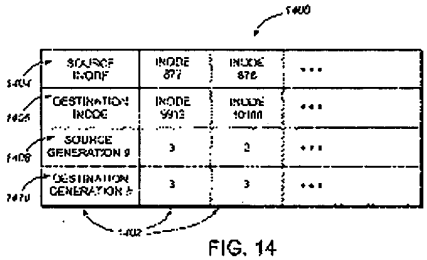


FIG. 14

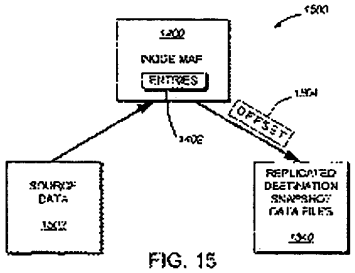


FIG. 15

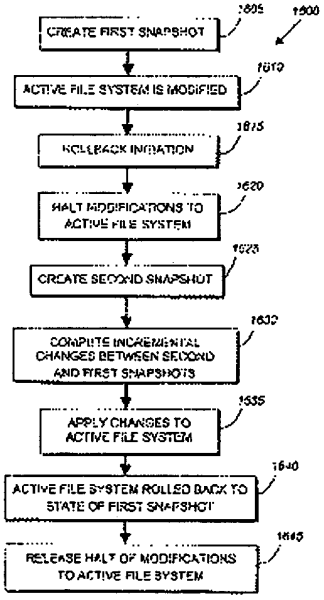


FIG. 16

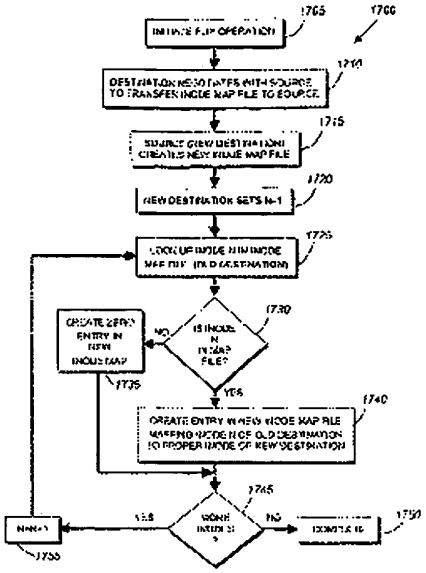


FIG. 17

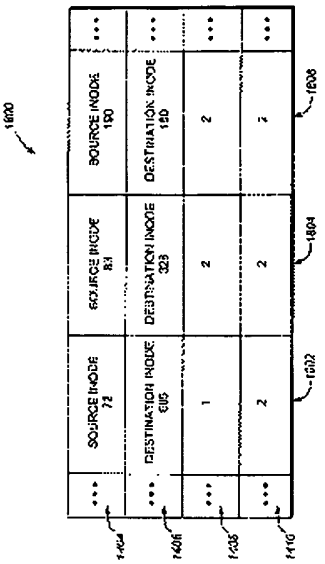


FIG. 18

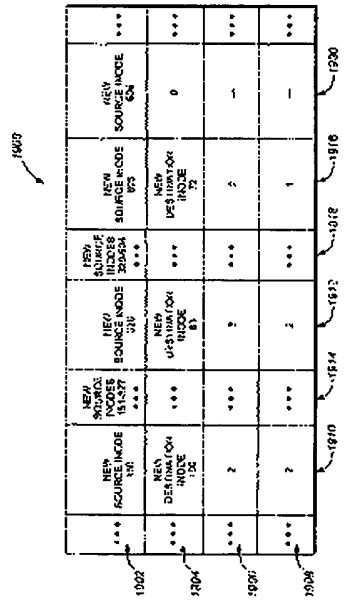


FIG. 19

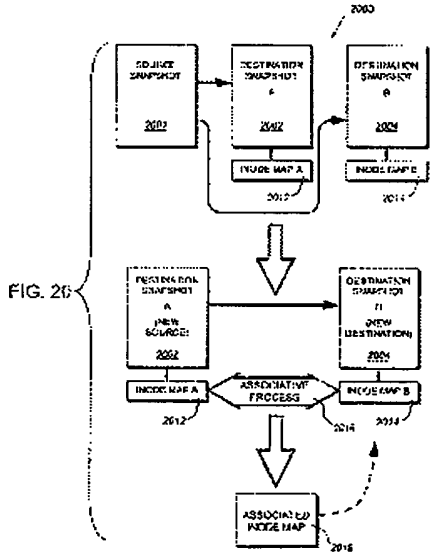


FIG. 20